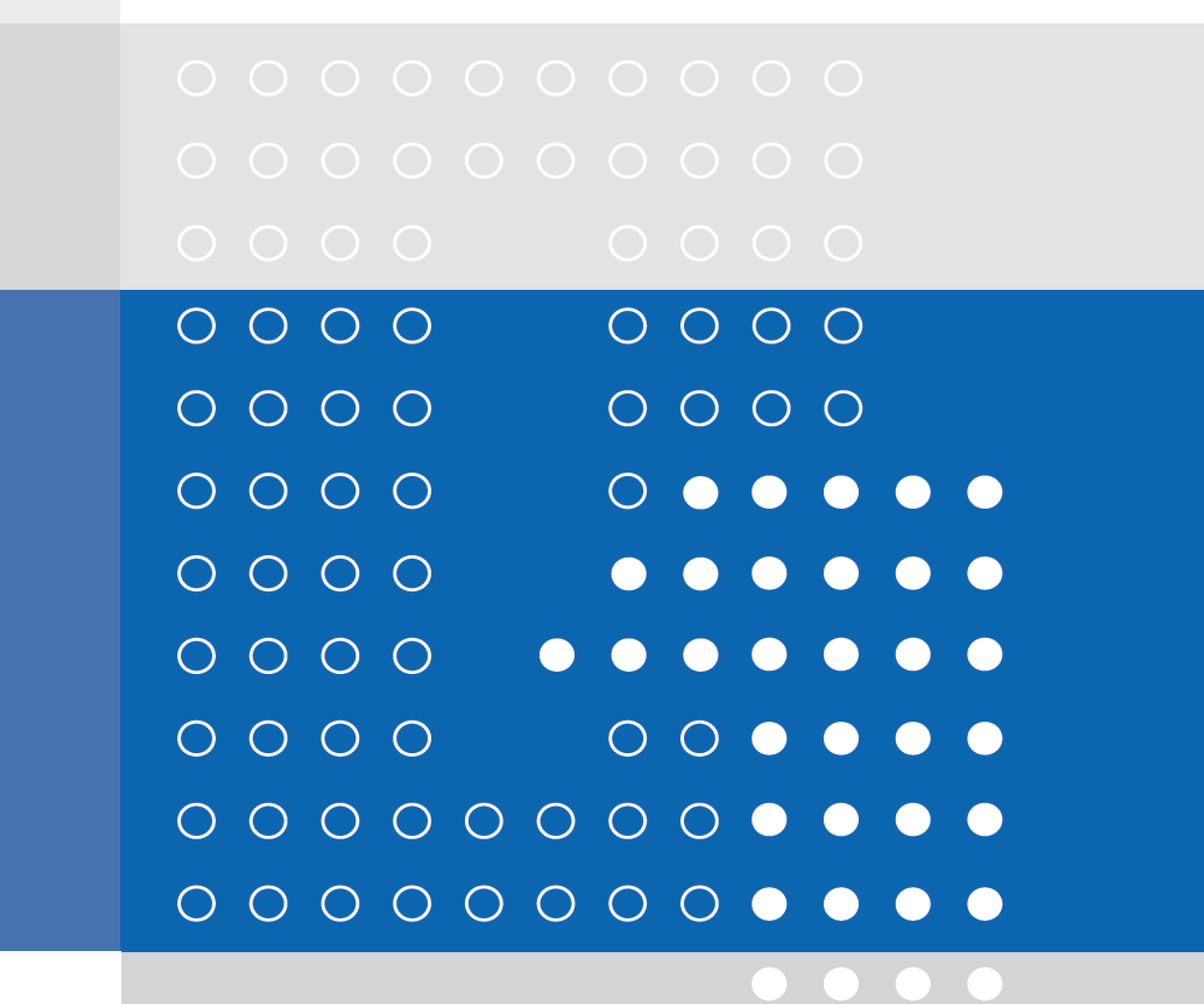
计算机系列教材

软件测试导论



蔡立志 编著

清华大学出版社

计算机系列教材

软件测试导论

蔡立志　编著

清华大学出版社
北　　京

内 容 简 介

本书系统地介绍了软件测试的主要技术和方法，全书共分为8章。在讨论黑盒测试时介绍了边界值、等价类、决策表、因果图等方法，而基于控制流的测试则介绍了语句覆盖、判定、条件覆盖、修正判定覆盖和基本路径覆盖；在组合测试方面包括拉丁方阵、正交表等方法，重点讨论了成对组合测试方法；基于有限自动机的测试涵盖了T方法、D方法、W方法、U方法；面向对象语言测试方面包括类属性、对象属性测试，基于对象创建和销毁、装饰器、多态的软件测试；基于UML的软件测试介绍了基于用例图、类图、活动图、序列图、状态图等的测试方法；最后，本书讨论基于Petri网的软件测试、蜕变测试、基于变异的软件测试以及基于故障树的软件测试。

本书结合当前技术发展的特点和工程实践的需求而编写，内容新颖，实操性强，既适合于软件测试和软件质量保障相关的技术人员在从事软件测试时参考，也适合作为软件工程专业的本科生、研究生学习软件测试课程的教科书。

图书在版编目(CIP)数据

软件测试导论/蔡立志编著. --北京：清华大学出版社，2016
计算机系列教材
ISBN 978-7-302-42821-3

Ⅰ. ①软…　Ⅱ. ①蔡…　Ⅲ. ①软件－测试－教材　Ⅳ. ①TP311.5

中国版本图书馆CIP数据核字(2016)第028541号

责任编辑：白立军　薛　阳
封面设计：常雪影
责任校对：时翠兰
责任印制：何　芊

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：三河市春园印刷有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：24.5　　**字　　数**：568千字
版　　次：2016年8月第1版　　**印　　次**：2016年8月第1次印刷
印　　数：1～2000
定　　价：49.50元

产品编号：059067-01

《软件测试导论》序

每当人们回顾技术进展时,都会一致认同信息技术是发展最快的技术之一,特别是信息技术的渗透性,几乎在各个领域中都可以看到它的身影,从而使我们的世界变得更加精彩。软件作为信息技术的灵魂,更是扮演了极其重要的角色,在现代社会中已经很难想象没有软件会是什么样?所以软件产业正在全球经济中占据越来越重要的地位,而软件质量已成为软件产品健康发展的关键技术。从20世纪发生软件危机以来,软件工程在过程质量管理、软件产品质量管理中都得到了快速的发展。在软件过程质量模型方面,集成的能力成熟度模型CMMI是软件工程发展的一个重要标志,CMMI模型通过提升过程生产质量,从而提升软件产品质量。生产过程的控制依赖于严格的、规范的流程和文档得以实现。但是在实践中,这种重载的过程管理模式,并不适应需求的变化,所带来的效率负担和产品质量提升的关系并不能令人满意。于是又出现各种敏捷的开发过程模型,以期充分快速地适应需求的变化。在软件产品质量模型方面,也经历GB/T 16260—1996软件质量模型、GB/T 16260—2006软件质量模型、ISO/IEC 25010—2014软件质量模型。质量考察的对象从原来的"软件产品"扩展到"系统与软件工程"。无论是重载的规范模型还是轻量级的敏捷模型,或者哪种软件质量模型,软件测试都在其中发挥了无法替代的作用,也是软件产品发布前的最终检验。

随着软件产业的发展,软件测试的从业人员不断增加,对软件测试知识的学习需求也日益强烈。软件杀虫剂效应使得经典的软件测试方法日益失效,但是软件缺陷并没有因此而消失,无论软件开发技术怎样发展,软件缺陷始终像永远无法驱离的阴霾,潜伏在软件之中,新的开发模式和技术促使新的测试技术的发展。在这个背景下,作者结合该领域研究的最新成果和实践,倾注十余年的测试经验编著了本书。该书具有以下几个方面的特点。

1. 系统性

较为系统地介绍了软件测试的基本理论、基本方法和应用例子,涵盖了组合测试、基于有限状态机、面向对象结构测试、基于UML的测试等。

2. 新颖性

在介绍传统的黑盒测试和控制流测试的基础上,立足于当前国内外的最新研究成果,介绍当前测试领域发展的新趋势、新技术。例如,基于Petri网的测试、蜕变测试等。

3. 实用性

软件测试既是一门理论性较强的学科，更是一门实践性和应用性特别强的技术。作者既注重理论的探索，更注重实践的应用，对于每一个测试方法都给出了具体应用的例子，在条件许可的情况下，给出了运行的 Python 程序。

4. 简明性

对当前流行的软件测试技术做了深入浅出的阐述，但没有涉及自动化测试和系统信息安全测试等，读者就能简洁明了地掌握软件测试技术的核心内容。

在软件测试领域，还有很多值得探索的研究课题，例如，测试的 Oracle 问题、测试充分性问题等。这些课题的解决，尚需几代人的不断努力。2011 年，国务院文件([国发〔2011〕4 号])中第十八条指出："鼓励软件企业大力开发软件测试和评价技术，完善相关标准，提升软件研发能力，提高软件质量，加强品牌建设，增强产品竞争力。"将软件测试纳入国家软件产业发展的高度进行鼓励。

该书问世，将有益于读者掌握一门重要的技术，有益于提升我国软件产品的质量，有益于推动软件测试的研究、教学、生产的进一步发展，该书是一本值得推荐的优秀著作。

朱三元

2016 年 4 月

前 言

自从20世纪软件危机以来，软件工程得到了快速发展，日益受到学术界和产业界的重视。国际标准化组织ISO和国际电工委员会IEC第一联合技术委员会第七分技术委员会ISO/IEC JTC1/SC7，即软件和系统工程分技术委员会，专注于软件工程国际标准的研制。为了适应软件技术发展的需求，SC7成立了两个和软件测试密切相关的工作组WG6和WG26。WG26专注于研究软件测试标准化工作，2013年发布了ISO/IEC 29119《软件和系统工程 软件测试》系列标准，从概念与定义、测试流程、测试文档、测试技术4个部分加以阐述。WG6专注于软件质量的研究，2014年发布了ISO/IEC 25051《系统与软件工程 系统与软件质量要求和评价(SQuaRE) 就绪可用软件产品(RUSP)的质量要求和测试细则》标准。中国信息技术标准化委员会软件工程分技术委员会专门成立了软件质量与测试工作组，推进软件测试标准化的各项工作。2011年，国务院学位委员会"关于印发《学位授予和人才培养学科目录(2011年)》的通知"(学位[2011]11号)文件确定软件工程学为一级学科(080835)，标志着软件工程学科进入了一个新的发展阶段，是软件工程学科发展的一个重要的里程碑。在2000年国务院18号文《鼓励软件产业和集成电路产业发展的若干政策》(国发[2000]18号)的背景下，全国成立一批第三方独立的软件测试机构从事第三方软件测试业务，为创建良好的软件产业氛围做出了重要贡献。2005年，在北京成立了全国第三方软件评测机构联盟，截至2015年，成员单位由成立之初的十多家发展到2015年的五十多家，软件测试技术发展需求极其旺盛。

自从1946年出现第一个Bug以来，软件测试作为软件质量保证的重要手段之一，在软件工程领域发挥着越来越重要的作用。软件测试自身也逐渐发展成为相对独立的软件工程过程，包含测试策划、测试设计、测试实现、测试执行、测试分析等阶段。但是随着开发技术的发展，带来了日益明显的软件测试杀虫剂效应。工程技术人员将软件测试所发现的软件缺陷的共性特征加以归纳整理，以组件内在属性的方式出现。这些组件在应用之前都已经单独通过测试与验证，例如边界判定、电话号码、身份证的信息有效性验证。基于这些组件开发的软件产品，利用边界值分析、等价类划分等测试方法已经难以发现有价值的缺陷。另一方面，面向方面编程AOP技术使得核心业务和辅助功能分离成为可能，例如日志记录、参数有效性验证等独立于核心业务进行开发，进一步加剧了软件测试的杀虫剂效应，使得经典的测试技术发现软件缺陷的能力在日益降低。动态语言、函数式编程带来了软件编程全新的思维，Z语言、Petri网、UML等新型建模技术和工具应用日益广泛，都促使产业界和学术界不断探索和发展新的测试技术和方法。作者从事软件测

试十多年，越来越深刻地体会到软件测试的变化。为了适应软件技术发展的趋势，结合这些年来软件测试工作实践，将软件测试的工程实践和科学研究的成果编著成《软件测试导论》这本书，与从业者分享。

书中所有的程序代码都采用 Python 语言编写。Python 语言的简洁性，使得读者能够将注意力集中在测试的核心思想。在阐述测试方法和原理方面，努力避免受程序设计语言特性所影响，但非常遗憾，无法做到完全避免语言的相关性。例如，第 6 章面向对象结构的测试，若抛开特定语言，抽象的测试概念会降低读者的感性认识。尽管在不同程序设计语言中，面向语言特性的实现中存在一定的差异，例如，是否存在接口、是否支持多重继承等，第 6 章还是采用 Python 语言相关的特性来描述。在 8.3 节关于变异测试的内容中，许多变异规则是和静态语言特性相关联的，在 Python 语言中并不适用，为了不失一般性，本书描述了这些变异规则。

受篇幅所限，本书没有阐述包括软件质量保证过程、软件测试成熟度模型、软件自动化测试、软件系统的信息安全测试等内容，也不涉及某些特定对象测试，例如移动 APP 测试、Web 测试。这些内容和本书所讨论的软件测试存在密切的关系，但是这些内容都是独立的体系，都可以单独出版。本书仅涵盖当前流行的软件测试技术。

第 1 章，主要介绍软件测试的历史和发展、测试术语、软件缺陷管理、软件质量模型和测试，并探讨了软件测试的局限性及其分类。

第 2 章，主要介绍边界值、等价类、决策表、因果图等传统的黑盒测试方法。

第 3 章，在介绍图论、控制流图知识的基础上，以 Python 语言为例介绍了基于语句覆盖、判定、条件覆盖、修正判定覆盖和基本路径覆盖的测试方法。

第 4 章，在介绍基于拉丁方阵、正交表测试方法的基础上，重点介绍组合测试的要求及其测试用例的生成方法，讨论了可变强度组合测试以及约束对组合测试的影响。

第 5 章，介绍有限自动机的定义、特性以及故障模型，在此基础上，探讨了有限自动机的测试方法，包括 T 方法、D 方法、W 方法、U 方法。

第 6 章，以 Python 语言为例，介绍面向对象语言测试，包括类属性、对象属性测试、基于对象创建和销毁的测试，以及基于装饰器和多态的软件测试。

第 7 章，重点介绍基于用例图、类图、活动图、序列图、状态图的测试，对于每一类 UML 图都涵盖了概念、覆盖准则和测试用例设计三个部分。

第 8 章，重点介绍基于 Petri 的软件测试、蜕变测试、基于变异的软件测试以及基于故障树的软件测试。

本书第 2 章由郑阳和陆佳文协助撰写，第 7 章由陆佳文协助撰写，陆佳文、张杨、刘振宇等人协助对全文进行校对，感谢他们的真诚付出。

最后，感谢清华大学出版社在本书出版过程中付出的所有努力和帮助。感谢上海计算机软件技术开发中心的同事们，本书的出版离不开他们的共同努力。

由于作者才疏学浅，时间匆忙，书中难免存在疏漏或者不足，恳请读者批评指正。反馈意见请发邮件到 clz@ssc.stn.sh.cn。

蔡立志

2016 年 4 月

目 录

第1章　绪　　论

随着软件应用的日益深入和广泛，各个行业都已经离不开软件的支撑，软件质量问题引起的损失也日益增大。本章介绍了软件测试的发展以及历史上著名的软件质量问题。软件缺陷管理在整个软件生存周期中发挥着极其重要的作用。软件缺陷生存周期、软件缺陷的属性是软件缺陷管理的核心内容。软件质量模型经历了多个版本，为软件测试提供了有价值的依据。软件测试模型描述了软件测试和开发过程之间的关系。最后本章分析了软件测试局限性和分类。

1.1　软件测试的历史和发展

1.1.1　软件测试的起源

在软件测试领域，软件的缺陷通常被称为 Bug。Bug 的意思是小虫。为什么将软件缺陷称为 Bug？有一段历史典故。1946 年，Grace Hopper 从职位退休以后，加入了哈佛大学的计算实验室，继续 Mark Ⅱ和 Mark Ⅲ的研究工作。1947 年 9 月 9 日，Mark Ⅱ计算机遇到一个错误。经调查发现这个错误是由于 F 面板(Panel F)上第 70 号继电器中间飞入了一个飞蛾而导致电路信号错误。这个飞蛾被贴在了日志上，日志上写着：“First actual case of bug being found.”，如图 1-1 所示，这就是软件缺陷的起源。该日志目前保存在史密森尼博物院中。

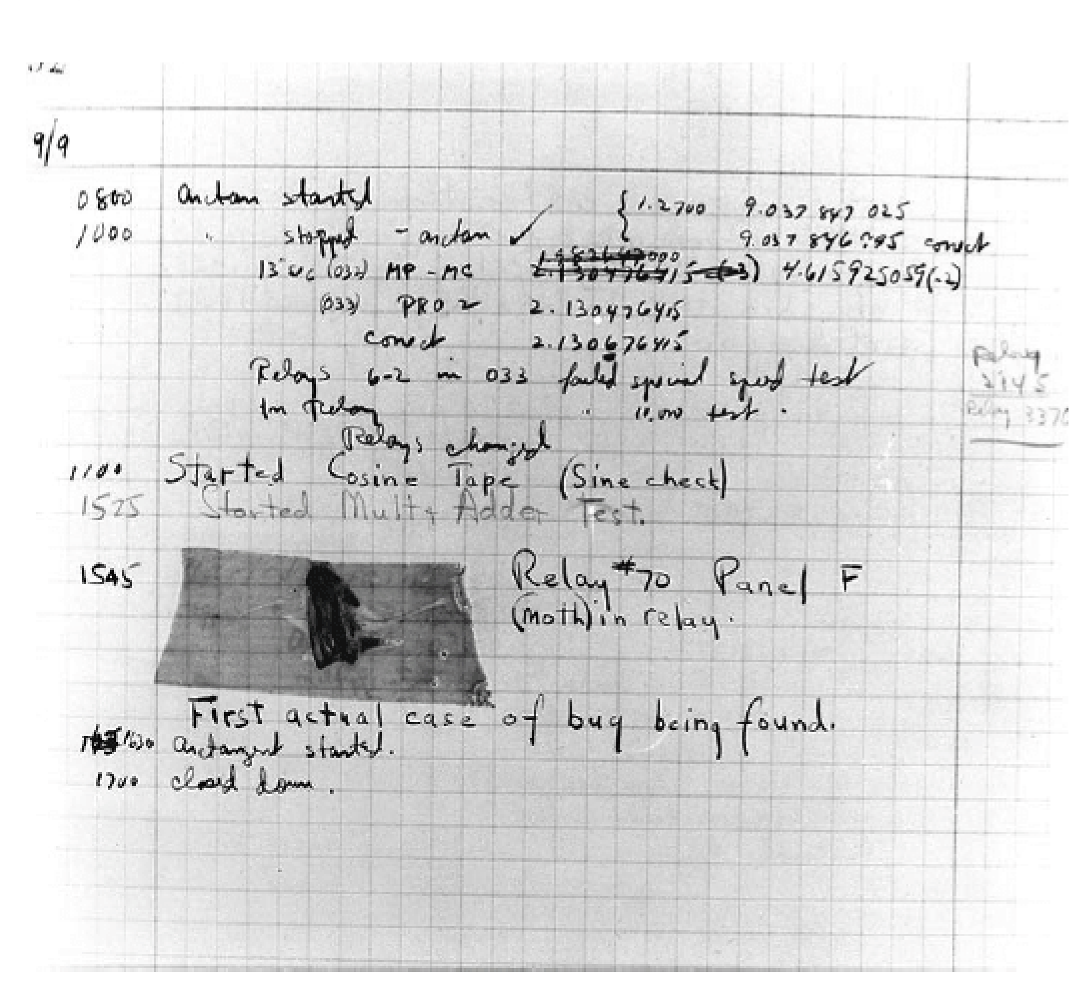


图 1-1　Mark Ⅱ上的 Bug

早期的软件是用机器语言编写的，机器语言是内置在计算机电路中的指令，由 0 和 1 组成。不同的计算机使用不同的机器语言，程序员必须记住每条语言指令及其二进制数

字组合。后来在机器指令的基础上出现了汇编语言，使用助记符表示每条机器语言指令，例如ADD表示加，SUB表示减，MOV表示移动数据。汇编语言需要大量的记忆指令，并且在表达复杂逻辑方面的能力非常弱，通常程序的功能也非常有限。

在那个时期，软件规模一般都很小、复杂程度低，没有形成成熟的软件开发过程体系，测试的含义比较狭窄。一般将测试等同于“调试”，目的是纠正软件中已经知道的故障，常常由开发人员自己完成这部分工作。

软件缺陷(Defect)是程序本身存在的问题，是程序原来就具有的。“缺陷”一词更能反映事情的本质，因为Bug是从外面爬进去的，是外部因素，并非程序本身存在的问题。

1.1.2 软件质量问题

随着软件技术不断发展，软件在各个领域都得到了非常广泛的应用，其作用也越来越重要。软件由于存在缺陷而导致的损失也越来越大，历史上曾经出现过几个著名的软件质量事故案例，涉及航空、航天、金融、工业控制、电子商务等各个领域，每一个质量事故都造成了重大的经济损失。

1996年6月4日，欧洲阿丽亚娜5型运载火箭的发射系统代码直接重用了4型的相应代码，而4型的飞行条件和5型的飞行条件截然不同。软件引发的问题导致火箭在发射39s后偏轨，从而激活了火箭的自我摧毁装置，此次事故造成了3.7亿美元经济损失。

1999年，NASA在制造其火星气候轨道探测器时，使用的是英制单位，而不是预定的公制单位，由于测试不充分导致该缺陷未被及时发现。该错误造成探测器的推进器无法正常运作，探测器从距离火星表面130英尺的高度垂直坠毁。此项工程成本耗费3.27亿美元。

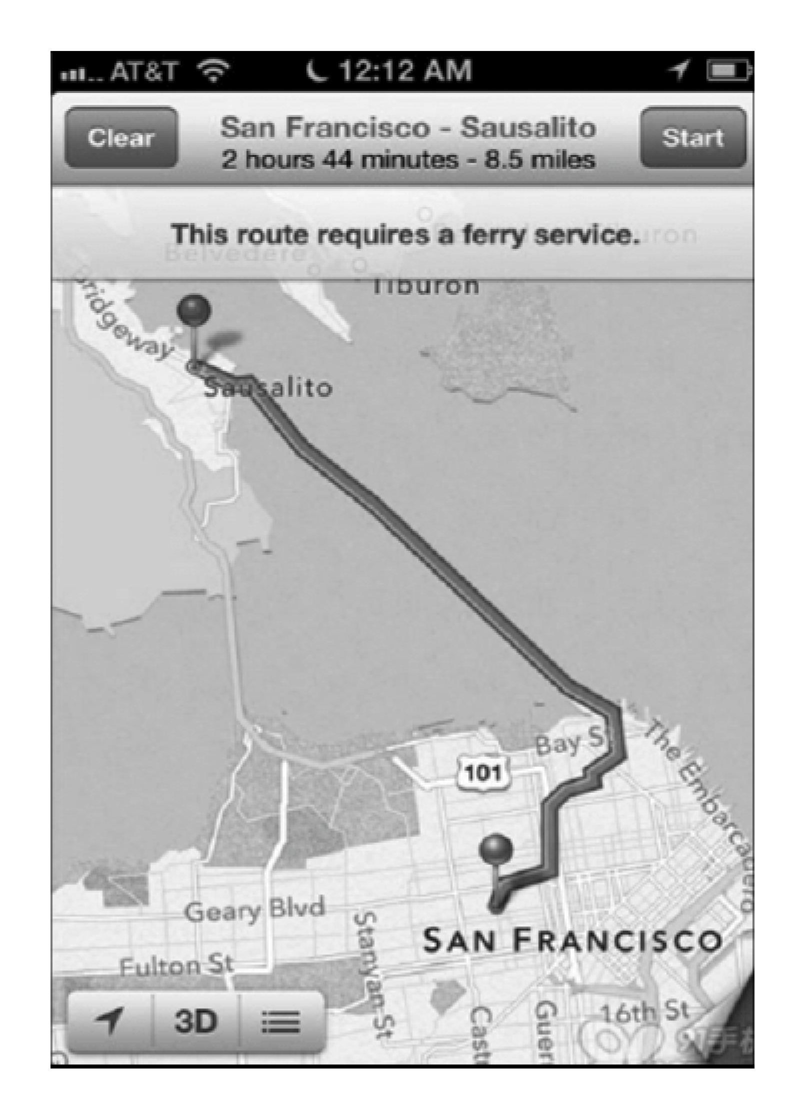
2002年6月28日，美国国立标准技术研究所(National Institute of Standards and Technology，NIST)发表的有关软件缺陷的损失调查报告指出：据推测，由于软件缺陷而引起的损失额每年高达595亿美元。

2006年，美国国税局IRS因技术人员对程序进行重新设计，导致电子诈骗系统不能正常运行。该错误直接带来的经济损失达两三亿美元，修复该错误的成本高达2100万美元。

2012年，美国KCP(Knight Capital Group)金融公司由于电子交易系统出现故障，交易算法出错，导致该公司对150支不同的股票高价购进、低价抛出，直接给公司带来了4.4亿美元的损失，当天股票下跌62%。

2012年，苹果iOS 6发布，由于缺乏充分的测试，其内置的地图服务存在许多地点和定位的错误，如图1-2所示。这些错误导致1000万用户在48个小时内纷纷涌向Google地图。

2014年，由于软件设计上的缺陷，黑客组织“蜻蜓Dragonfly”对石油管道运营商、发电企业和其他能源工控设备提供商发起攻击。在18个月内，全球有84个国家的工业控制系统受到了攻击，1018座发电站感染恶意程序。



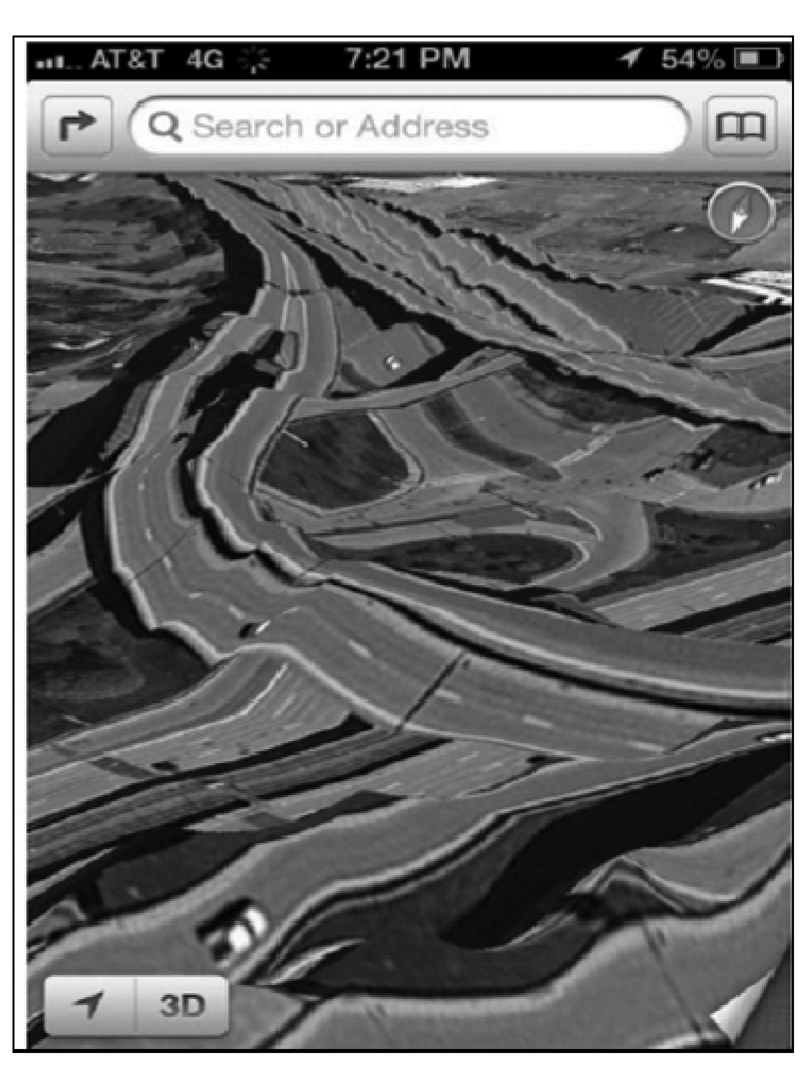


图 1-2　苹果地图错误

1.1.3　软件测试的发展

从 Mark Ⅱ 上出现的第一个缺陷(尽管这个缺陷不是人为造成的)开始,软件测试经过近七十年的发展,详细信息见附录 A。在这个过程中,形成 4 种不同的软件测试观点。

1. 软件测试等同于调试活动

这种观点存在于早前的阶段,认为软件测试的目的是定位或者纠正软件中存在的故障。一般情况下,软件项目中没有独立的测试人员,执行软件测试任务的是软件开发人员。软件测试没有形成独立阶段,也不是独立的任务。

但是软件测试和调试还是存在很大区别。调试的基本手段包括在程序中设置断点、观察内容变量的变化、逐步跟踪程序的执行等。尽管在调试时,也会设计一定的输入参数,但是其重点在于是否能够定位已经发现缺陷的原因,而不会考虑软件全面的特性。表 1-1 给出了软件测试和调试之间的比较。

表 1-1　软件调试和测试的比较

	软件测试	调　试
相同点	关心软件参数的输入	
	和软件故障相关联	
	测试发现的缺陷,可以成为调试的依据	
	依赖于软件运行	

续表

	软件测试	调 试
目的	尽可能发现软件缺陷，或者确认软件和预期的符合性	定位软件已经发现的故障原因
用途	为软件开发过程改进提供依据	为纠正故障提供依据
人员	一般由独立的测试人员执行	由开发人员执行
阶段	一般有独立的软件生存周期阶段	一般和开发过程融合在一起
技术	等价类、边界值、组合测试……	断点、跟踪、单步执行……

2. 软件测试的目的是验证软件是否符合预期(求真)

1951 年，美国著名的质量管理大师朱兰(J. M. Juran)提出了质量控制手册，认为质量就是产品的适用性，是满足用户需求的程度。受该思想影响，软件测试和调试分离以后，在很长一段时间内，软件测试的目的被理解为"确信产品能够工作"，认为软件测试是证明软件是符合预期的。这种观点常见的表述包括：

(1) 软件测试是表明软件没有错误的过程(Testing is the process of demonstrating that errors are no present)。

(2) 软件测试的目的是表明程序正确执行了预期的功能(The purpose of testing is to show that a program performs its intended functions correctly)。

(3) 测试是建立一种信心，认为程序能够按预期的设想运行(Testing is process of establishing confidence that a program does what it is supposed to do)。

软件测试就是验证软件功能和需求是一致的，就是针对软件系统的功能点，逐步验证其正确性。这种观点的代表人物是 Bill Hetzel 博士，他在 1972 年组织了世界上第一次软件测试会议。1973 年，Bill Hetzel 博士给出了软件测试的定义：建立一种信心，认为程序能够按预期的设想运行。1983 年，他将软件测试的定义修正为：评价一个程序和系统的特性或能力，并确定它是否达到预期的结果。

这种观点存在一定的现实意义，例如第三方独立的软件测试机构所执行的验收测试，目的是验证软件和用户需求之间的一致性。这类测试以需求和设计、开发合同为基本出发点，设计测试用例，其目标比较明确。

3. 软件测试的目的是尽可能发现软件中的错误(证伪)

另一种观点认为软件测试的目的是尽可能多地发现软件中包含的错误。这种观点是基于一种假设，即软件本身包含错误。这种观点常见的表述包括：

(1) 测试是为了发现错误而执行程序的过程(Testing is the process of executing a program with the intent of finding errors)。

(2) 测试是发现程序中的错误并使得其成为一个可行的任务(Testing is the process of uncovering errors in a program that makes it a feasible task)。

(3) 测试是一个发现程序错误(假设程序中存在错误)的破坏性过程(Testing is a

destructive process of trying to find the errors (whose presence is assumed) in a program)。

这种观点的代表人物是Glenford J. Myers,他从心理学角度指出,如果以“验证软件是工作”为目的,将导致严重问题,并将其描述为“本末倒置的定义”。Glenford J. Myers在《测试的艺术》(*The Art of Software Testing*)一书中,如此描述:“人类的行为总是倾向于具有高度目标性,建立一个正确的目标有着重要的心理学影响。如果我们的目的是证明程序中不存在错误,那就潜意识中倾向于这个目标,也就是说,人们会倾向于选择可能较少导致程序失效的测试数据”。基于此,Glenford J. Myers给出了软件测试的三个重要观点:

(1) 测试是为了证明程序有错,而不是证明程序正确。

(2) 一个好的测试用例在于它能发现至今未发现的错误。

(3) 一个成功的测试是发现了至今未发现的错误的测试。

在这种观点的支持下,必然要求测试人员发挥主观能动性,使用逆向思维,以发现软件中存在的缺陷。

4. 质量保证和缺陷预防

上述的两种观点,在实际的应用过程中,都存在着一定的不足。随着软件不断向大型、复杂化发展,软件测试作为软件质量保证的重要手段,发挥了越来越重要的作用。软件测试的结果除了应用于缺陷修复外,同时反馈到公司产品前期的开发过程中,形成闭合的反馈机制,为缺陷预防提供参考。集成能力成熟度模型CMMI包含这些机制。CMMI的重要思想是通过改进软件开发过程质量从而提升软件产品质量。在CMMI中验证(Verification)和确认(Validation)两个过程域,和软件测试存在密切的关系。验证通过提供客观证据对规定要求已得到满足的认定,确认通过提供客观证据对特定的预期用途或应用要求已得到满足的认定,验证注重“过程”,确认注重“结果”。同时,度量和分析过程域的目的是开发和维持管理系统的度量,包含以下两个基本的步骤。

(1) 收集过程数据。

(2) 对数据进行分析,发现趋势和问题。

软件测试为过程度量和改进提供了有价值的数据,包括缺陷密度、缺陷关闭缺失、缺陷发布缺失、产品部件之间的缺陷分布等。

1.2 软件测试术语

在软件工程领域,存在大量与软件测试相关的术语。这些术语在日常的工作中,经常被误用或者错误理解,导致不同人员之间的沟通和交流存在很大的障碍。在开始讨论测试技术之前,下面先介绍部分重要的软件测试术语。

1. COTS 软件产品

2010年,我国发布了国家标准GB/T 25000.51—2010《软件工程 软件产品质量要

求与评价SQuarRE商业现货(COTS)软件产品的质量要求和测试细则》,该标准等同采用国际标准ISO/IEC 25051:2006 Software engineering—Software product Quality Requirements and Evaluation(SQurRE)-Requirements for quality of Commercial Off-The-shelf(COTS) software product and instructions for testing。GB/T 25000.51—2010给出了COTS软件产品的定义:由市场驱动的需要而定义的、通过商业方式提供的、其适用性已经得到各类商业用户证实的商业现货软件。

COTS软件产品包括:

- 产品说明(包括全部封面信息、数据表、网站信息等);
- 用户文档集(安装和使用该软件必需的);
- 包含在计算机中可感知的媒体(磁盘、只读光盘、可下载的互联网,等等)上的软件。

该定义改写自GB/T 18905.4—2002。

软件主要由程序和数据组成。

该定义也适用于可作为分别研制的物品而生产或支持的产品说明、用户文档集和软件,但对于典型的商业费用和许可考虑,该定义也许不适用。

2. RUSP就绪即用软件产品

随着Web服务以及云计算的不断发展,企业逐渐由提供软件产品向提供软件服务转型,以交付产品licence为目的的产品正在大量消失。商业现货软件的限定词已经无法满足产业对于测试的需求。笔者积极向国际标准建议,修正ISO/IEC 25051标准的适用范围,有幸被国际标准所接受。2014年,国际标准ISO/IEC 25051—2014 Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—Requirements for quality of Ready to Use Software Product(RUSP) and instructions for testing发布,将商业现货软件(COTS)修正为就绪即用软件产品(RUSP)。和ISO/IEC 25051:2014对应的国家标准,目前也正在修订中。

ISO/IEC 25051:2014将RUSP就绪即用软件产品定义为:无论是否付费,任何用户可以不经历开发活动就能获得的软件产品。

就绪即用软件产品包括:

- 产品说明(表现形式可以包括封面信息、数据表、网页信息等);
- 用户文档集(安装和使用软件所必需的文档),包括操作系统的配置、软件产品所要求的目标计算机;
- 计算机媒介上的软件(磁盘、CD-ROM、网络下载等)。

软件主要由程序和数据组成。

本定义也适用于产品说明、用户文档集,以及作为单独的制成品而被生产和支撑的软件,该软件不收取通常的商业费用和证书费用。

上述定义可以看出,软件产品由产品说明、用户文档集、软件所构成。对于一个软件产品的测试包含三个部分,作为本书而言,所讨论的仅是软件,也就是程序和数据。

3. 软件测试的目的

2008 年，国家标准 GB/T 15532—2008《计算机软件测试规范》发布，该标准阐述了计算机软件测试的目的：

验证软件是否满足软件开发合同或者项目开发计划、系统/子系统设计文档、软件需求规格说明书、软件设计说明和软件产品说明等规定的软件质量要求；

通过测试，发现软件缺陷；

为软件产品的质量测试和评价提供依据。

从这个定义上看，软件测试的目的融合求真和证伪两个观点，软件测试既要关注软件和需求之间的符合度，同时要尽最大努力去发现软件中存在的缺陷。

4. 错误 error

错误 error 指产生错误结果的人为行动。[ISO/IEC 24765:2009]

在软件生存期的各个阶段，都贯穿着技术人员直接或间接的活动，都可能由于技术人员不小心或者其他原因，在软件中引入非预期的内容。错误是指在软件工程活动中人员不希望或不可接受的行为，该行为结果是导致软件缺陷的产生。因此错误是一种人为过程，相对于软件而言，错误是一种外部行为。

5. 缺陷 defect

工作产品中的瑕疵或缺点，使得该产品不能满足需求或者规格说明，并且需要修复或者替换。[GB/T 25000.1—2010]

软件缺陷存在于软件中，其结果是软件运行于某一特定条件时，会出现软件故障，这时称软件缺陷被激活。缺陷是人为错误所产生的一个包含在软件中间的结果，缺陷依附在软件中间。

6. 故障 fault

故障 fault 是指软件中差错的表现。[ISO/IEC 24765:2009]

软件故障是一种动态行为，是当软件遇到缺陷时的一种表现。软件有缺陷，不一定会产生故障。软件运行没有遇到缺陷，那么故障就不会出现。

7. 失效 failure

失效 failure 是指产品执行所需的功能被终止或者产品不能够在规定限制内执行成功。[GB/T 25000.1—2010]

当软件遇到故障，若此时没有适当的措施（例如容错）加以处理，便产生软件失效。

8. 问题 problem

问题 problem 是指由于系统使用过程中的不理想，导致一个或多个人遇到的困难或不确定性。

错误、缺陷、故障、失效、问题是软件测试领域中使用最频繁的术语。它们之间相互关

联，又存在很大的差异，由 IEEE 软件与系统工程标准委员会起草的 IEEE Std 1044—2009《软件异常分类》给出了不同定义之间的逻辑关系，如图 1-3 所示。

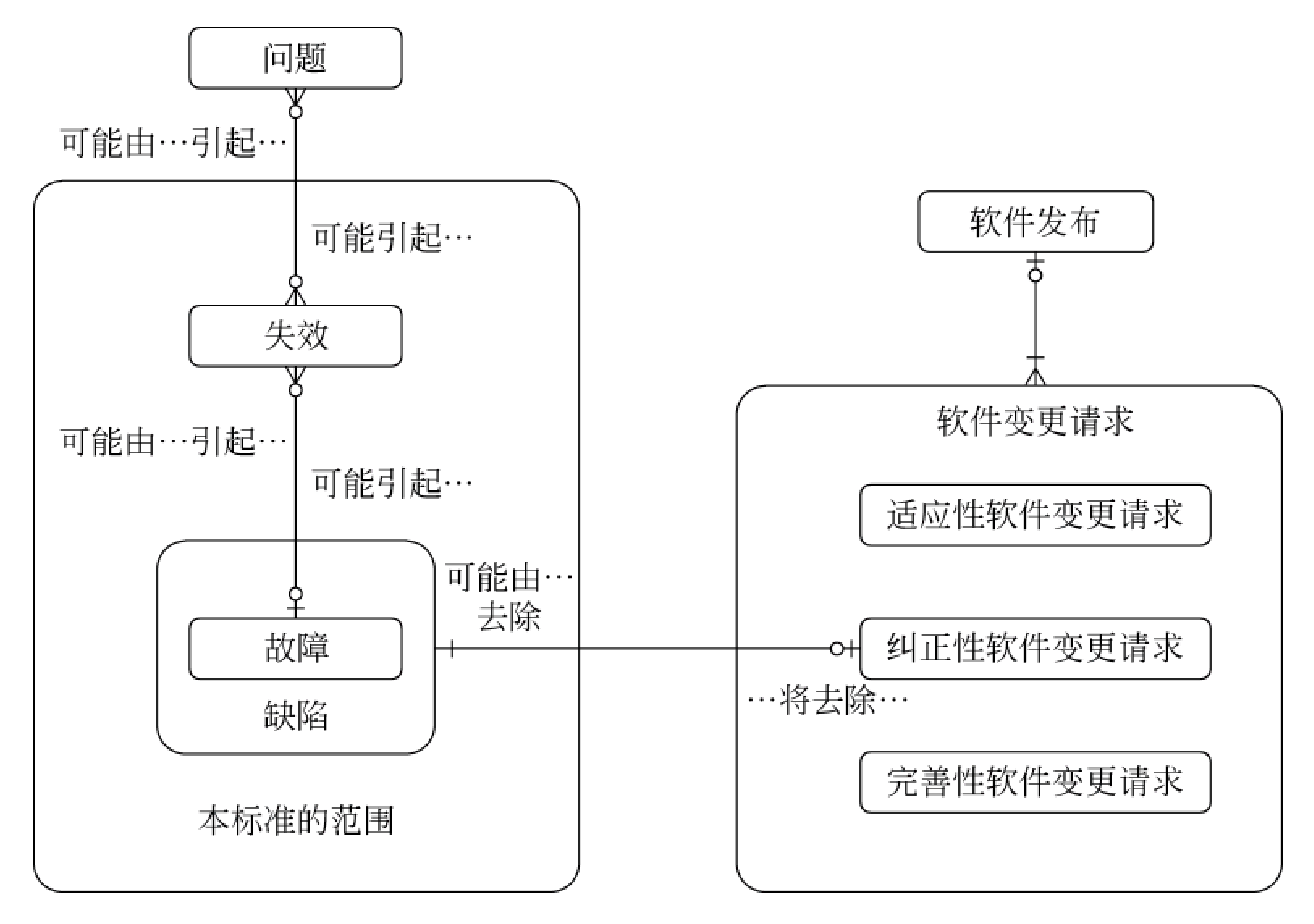


图 1-3 软件缺陷及其相关术语

问题可能由失效引起，而失效由故障引起，故障是缺陷的一种子集。缺陷由软件变更进行解决。错误的主体是人，编程人员在程序编码时，由于编程技术水平、精神状态、健康状态等原因，往往会发生错误。例如在输入代码时，不小心将加号＋误输入成减号－，或者在 C 语言编程中混淆位运算 & 和逻辑运算 && 等。在 Python 语言中，位运算和逻辑运算完全区分，减少了这种可能。位运算为 &（按位与）、|（按位或）、^（按位异或）；而逻辑运算为 and（逻辑与）、or（逻辑或）、not（逻辑非）。

软件各种异常之间的关系如表 1-2 所示。

表 1-2 不同软件异常之间的关系

类/实体（对）	关系描述
问题-失效	一个问题可能由一个或多个失效引起。 一个失效可能引起一个或多个问题
失效-故障	一个失效可能由一个故障引起。 一个故障可能引起一个或多个失效
故障-缺陷	故障是缺陷的子类。 故障都是缺陷，但缺陷不都是故障。 如果缺陷在软件执行期间出现（引起了失效），该缺陷就是一个故障。 如果通过检查或者静态分析发现缺陷并且在软件执行前将其消除，该缺陷就不是故障
缺陷-变更请求	缺陷可以通过纠正性变更请求来消除。 纠正性变更请求的目标是消除缺陷。 （变更请求最初用于适应性维护和完善性维护）

IEEE Std 1044—2009《软件异常分类》对应的国家标准已经列入修订计划，目前笔者正在进行修订。

1.3 软件缺陷管理

1.2节讨论了软件缺陷以及相关术语，软件缺陷作为软件测试的最重要资产之一，在软件质量保障过程中发挥极其重要的作用。测试执行过程中，发现软件失效后，提出书面的报告，提供给开发人员或者其他负责人员作为定位缺陷的依据，作为企业过程改进的度量数据依据。

在软件生存周期的不同阶段，都可能在软件中引入缺陷，如需求规格说明书描述的错误或者逻辑上的矛盾，设计和需求规格说明书不一致等。和需求、设计相关的缺陷，可以通过评审或者其他质量保障活动来发现，本书主要关注软件中的代码实现相关的内容。

常见的软件缺陷表现形式有以下几种。

(1) 软件出现的运行错误。

(2) 软件未达到产品说明的功能。

(3) 软件出现产品说明书指明不会出现的错误。

(4) 软件功能超过产品说明书指明的范围。

常见的缺陷用途包括：

(1) 作为修复缺陷的基本输入和依据。

(2) 缺陷的关闭趋势，在一定程度上表征了软件产品的稳定性。

(3) 缺陷的模块分布，有助于企业分析影响软件质量的因素。

(4) 缺陷的团队分布，有助于企业分析不同团队的管理模式的效果。

(5) 缺陷的严重程度分布，有助于企业做出产品发布的决策。

1.3.1 缺陷生存周期

软件测试发现了软件中存在的缺陷以后，软件缺陷开始进入其生存周期的管理。图1-4给出了软件缺陷的生存周期的示意图。

在软件缺陷的生存周期内，各个状态的含义如下。

(1) 新建(New)：新提交的软件缺陷。

(2) 公开(Open)：还没有解决且处于公开状态的缺陷，正在等待处理。

(3) 重新公开(Reopen)：通过验证软件缺陷处于未修复状态，需要重新公开。

(4) 分配(Assigned)：该缺陷已经分配给特定的人员。

(5) 修正(Fixed)：已完成修正，等待测试人员验证。

(6) 拒绝(Declined)：录入错误的，或讨论后决定不予处理的缺陷为拒绝状态。

(7) 延期(Deferred)：有争议的或者由于技术原因暂时无法解决的，可以将其设置为延迟处理状态。

(8) 关闭(Closed)：缺陷已被修复，测试人员验证后，确认缺陷处于关闭的状态。

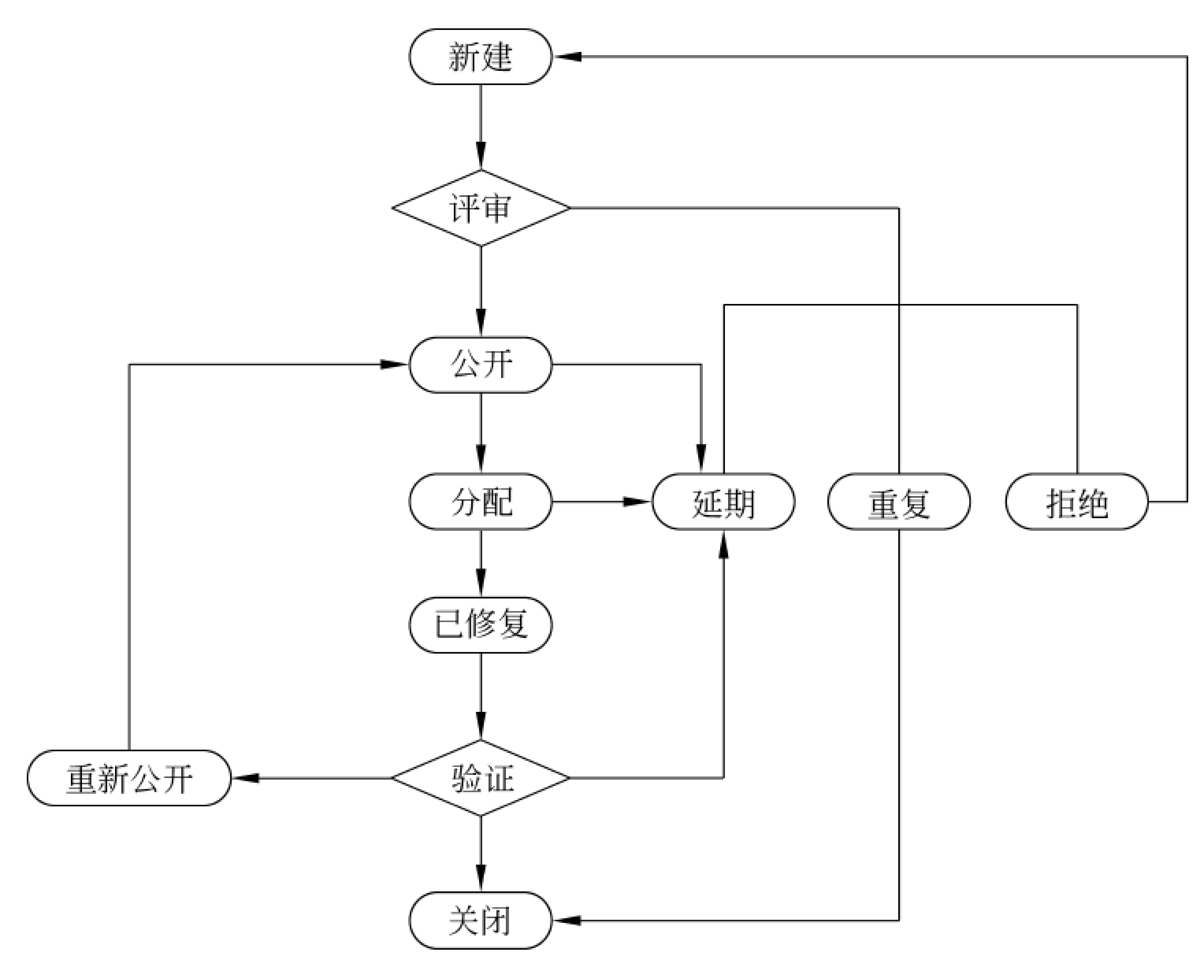


图 1-4　软件缺陷生存周期

当测试工程师或者用户发现软件缺陷以后，向缺陷管理系统提交该缺陷时需详细描述缺陷的属性。缺陷的属性包括静态属性和动态属性。静态属性在缺陷的生存周期内不会发生变化，例如缺陷的描述、缺陷的发生频率、产生缺陷的详细过程等。动态属性的值随着过程会发生变化。当该缺陷在缺陷管理系统创建以后，标志着该缺陷的诞生，其状态为 New，该缺陷开始进入缺陷管理。企业对处于 New 状态的缺陷进行审核，如果通过评审，缺陷将进入公开状态(Open)。若提交缺陷已经在缺陷管理系统中存在，那么新提交的缺陷直接进入关闭状态(Closed)。或者新提交的缺陷，经确认不是一个真正的缺陷，那么该缺陷将被拒绝(Declined)，若缺陷描述不恰当，缺陷也要被拒绝，测试工程师需要修改以后重新提交。或者，由于企业总体策略上的安排，该缺陷暂时不公开，则其进入延期状态(Deferred)。已经公开的缺陷，需要分配到开发工程师，进行缺陷修复。开发工程师完成修复，但是未经过测试，缺陷处于修正状态(Fixed)。测试工程师对已经修复的缺陷进行测试，若通过测试，该缺陷进入关闭状态。若通过测试发现，还存在该缺陷，那么进入重新公开状态(Reopen)。在这个过程中，公开、分配或者验证都可能会进入延期状态。

1.3.2　缺陷的描述及属性

当测试发现一个缺陷以后，提交新发现的缺陷，就进入缺陷管理的第一个步骤。软件缺陷的描述是软件缺陷报告的基础部分，是测试人员与开发工程师交流的重要渠道。软件缺陷描述需要使用简单的、准确的、专业的语言。一般而言，缺陷描述需要遵循以下原则。

(1) 可重现性和可重复性：在缺陷的详细描述中提供精确的操作步骤；重复性是指

相同人员在同一条件下能够得到相同的结果,重现性是指不同技术人员或不同实验室在各自的条件下得到相同的结果。

(2) 定位准确:缺陷描述准确,不会引起误解和歧义。

(3) 描述清晰:对操作步骤的描述清晰,易于理解,应用客观的书面语,避免使用形容词或者带有主观判断的词语,要详细描述操作的具体步骤。

(4) 完整统一:提供完整、前后统一的软件缺陷的步骤和信息,按照一致的格式书写全部缺陷报告,例如菜单的描述方式、命令行的描述、操作的粒度等保持一致。

(5) 短小简练:通过使用关键词,可以使问题摘要的描述短小简练,又能准确解释产生缺陷的现象。

(6) 特定条件:许多软件缺陷在某种特定条件下才会表现,软件缺陷描述需要重视特定的细节(如操作系统、浏览器或某种设置等),以辅助开发人员找到产生缺陷的原因。

本节重点介绍缺陷的优先级和缺陷的严重程度两个属性。

缺陷的优先级:依据项目的总体目标和企业的业务策略,对于缺陷是否修复所做的一种分类。在不同的公司中,对于缺陷的优先级的设置会有较大的差异,应根据公司的目标做出决定。

(1) 紧急的(Urgent):需立即修复。

(2) 关键的(Critical):需要优先考虑。

(3) 主要的(High):需要修复的。

(4) 中等优先级(Medium):在满足特定条件下,可以暂时不用修复。

(5) 低优先级(Low):可以暂时不用修复。

缺陷的严重程度:软件缺陷对软件质量的破坏程度,即此软件缺陷的出现可能造成的损失程度。

(1) 致命:软件的意外退出甚至操作系统崩溃,死循环,数据库发生死锁,因错误操作导致的程序中断,数据通信错误,导致测试无法继续执行,可能影响其他模块功能。

(2) 非常严重:程序错误、程序接口错误,数据库的表、业务规则、默认值未加完整性等约束条件,关键功能完全不能实现,程序运行不稳定。

(3) 严重:软件的单个功能失效,操作界面错误,输入限制未放在前台进行控制,删除/退出操作未给出提示,功能不完整,如菜单、按钮不响应,对错误没有处理信息。

(4) 一般:界面不规范,辅助说明描述不清楚,输入输出不规范,提示窗口文字未采用行业术语,可输入区域和只读区域没有明显的区分标志。

(5) 建议:如 Tab 键跳转不正常;窗口中的按钮或者控件缺少快捷字母,或快捷字母冲突;文字表述中有错别字或歧义;测试人员所提出的建设性意见。

缺陷的优先级和严重程度是缺陷两个最重要的属性,它们相互关联。一般地,严重程度高的软件缺陷具有较高的优先级,但是严重程度和优先级并不总是一一对应。例如有个严重的缺陷,但其发生的频度很低,目前的客户群并不受该缺陷的影响,这个缺陷的优先级就不一定非常高。优先级高的,严重程度并非一定高。例如,公司名字和软件产品商标一旦误用了,将会造成非常大的损失。这种缺陷虽然是用户界面缺陷,并不影响用户使用,但影响公司和产品形象,也可能具有非常高的优先级。

除了缺陷的优先级和严重程度以外，软件缺陷属性还包括缺陷标识、缺陷类型、缺陷产生可能性、缺陷状态、缺陷来源、缺陷原因等。

1.4 软件质量模型的发展

在 GB/T 6583—1994《质量管理与质量保证术语》中对质量的定义：反映实体满足明确或者隐含的需要的能力的特性的总和。

软件质量模型从不同的视角给出了对于软件的基本要求，反映用户、供方、需方表达软件质量的重要手段之一。软件质量模型标准随着技术的发展，经历过三个大的版本，分别为 GB/T 16260—1996(等同采用 ISO/IEC 9126:1991)、GB/T 16260—2006 系列标准(等同采用 ISO/IEC 9126 系列)、ISO/IEC 25010:2014 等。

1.4.1 GB/T 16260—1996 软件质量模型

1996 年，我国发布了一个重要的软件质量标准 GB/T 16260—1996《软件产品评价——质量特性及其使用指南》。该标准等同采用国际标准 ISO/IEC 9126: 1991 Information technology-software product evaluation Quality characteristics and guideline for this use。GB/T 16260—1996 仅定义了 6 个特性，以最小的重叠反映软件质量，描述了对软件质量的不同观点、软件过程评价模型和评价步骤，但该标准没有定义子特性，而是将 21 个子特性作为参考信息列在附录中。

1.4.2 GB/T 16260—2006 软件质量模型

质量特性不仅可用于评价软件产品，而且也可用于定义质量需求以及其他用途，GB/T 16260—1996 发展成为两个相关的系列标准：GB/T 16260—2006 软件产品质量和 GB/T 18905—2002 软件产品评价。GB/T 16260—2006 系列标准将质量模型分为内部质量模型、外部质量模型以及质量模型。在 GB/T 16260—2006 中，内/外部质量模型中保留了 6 个软件质量特性，将子特性扩充为 27 个。同时增加了使用质量模型，包含 4 个特性，没有子特性。GB/T 16260—2006《软件工程　产品质量》分为如下几部分。

第 1 部分(即 GB/T 16260.1)：质量模型。

第 2 部分(即 GB/T 16260.2)：外部度量。

第 3 部分(即 GB/T 16260.3)：内部度量。

第 4 部分(即 GB/T 16260.4)：使用质量的度量。

图 1-5 给出了 GB/T 16260—2006 和 GB/T 18905—2002 标准之间的关系。

内部质量是基于内部视角的软件产品特性的总体，可以在代码实现、评审和测试期间被改进，但是内部产品质量的基本性质不会改变，除非进行重新设计。

外部质量是基于外部视角的软件产品特性的总体。在模拟环境中用模拟数据测试时，使用外部度量所测量和评价的质量，在测试期间，大多数故障都应该被发现和消除。

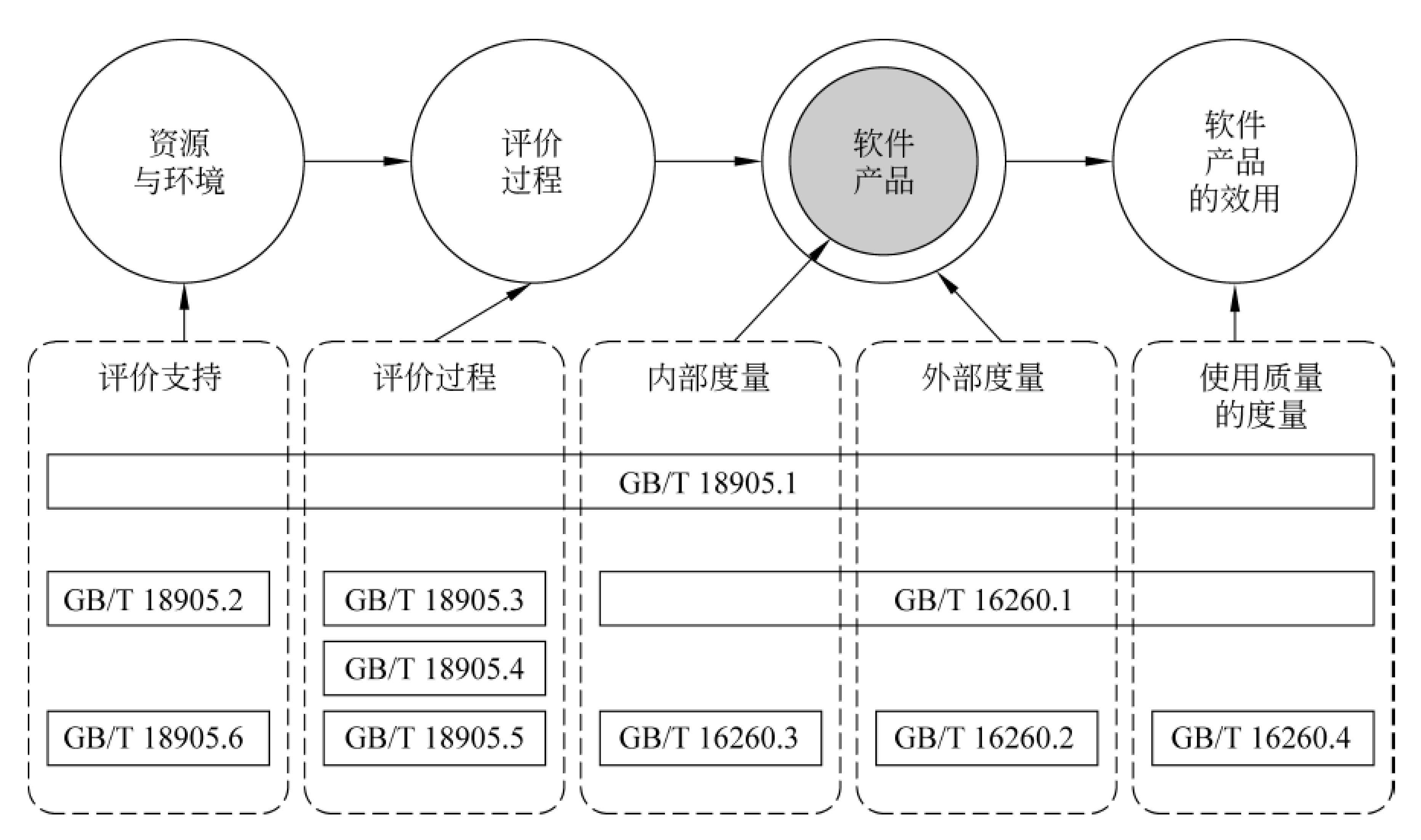


图 1-5 GB/T 16260—2006 和 GB/T 18905—2002 标准之间的关系

然而,在测试后仍会存在一些故障。

使用质量是基于用户观点的软件产品用于指定的环境和使用周境时的质量。它测量用户在特定环境中能达到其目标的程度,而不是测量软件自身的属性。使用质量有 4 个特性:有效性、生产率、安全性和满意度,没有子特性。

在 GB/T 16260—2006 中,外部和内部质量的质量模型在质量特性和子特性级别都是相同的,在具体的度量元上,存在不同的侧重点。内部质量和外部质量均包含 6 个质量特性:功能性、可靠性、易用性、效率、维护性和可移植性。每一个特性进一步细分为若干子特性,图 1-6 给出了 GB/T 16260—2006 定义的内/外部质量模型。用户可以选择、修改及应用其中的度量和测度,也可以针对独特的应用领域定义特定应用的度量。

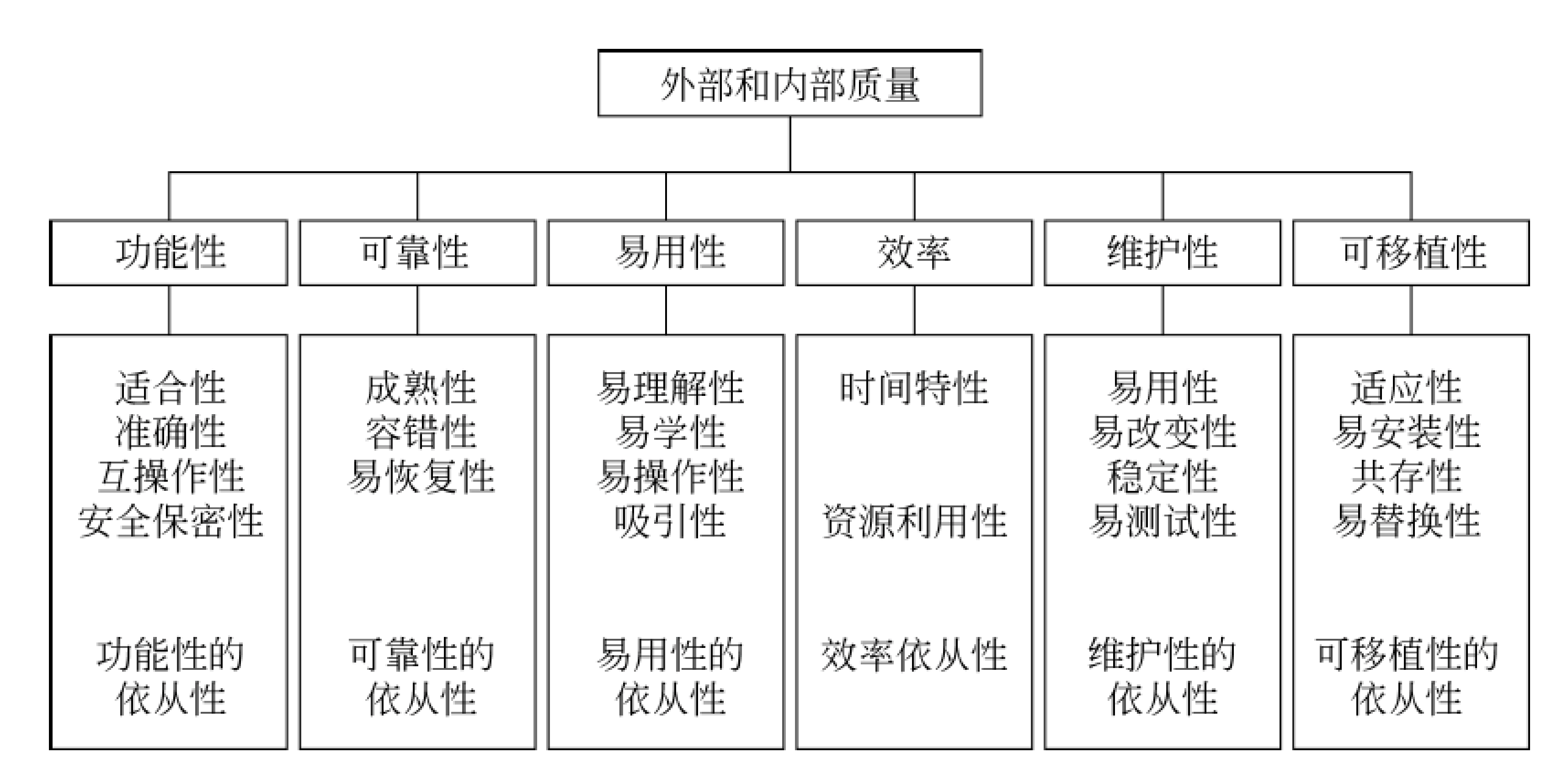


图 1-6 GB/T 16260—2006 定义的内/外部质量模型

GB/T 16260—2006 对度量元的定义存在一些不足。表 1-3 给出了 GB/T 16260—2006 部分度量元示例,不同的度量元之间存在很大的差异:有些包含单位,例如响应时

间;有些没有单位,例如功能的充分性;有些度量元的测量值越大越好,例如吞吐量;有些度量元的值越小越好,例如估计潜在的故障密度。这些差异使得用户在评价软件总体质量时存在很大的困难。

表 1-3　GB/T 16260—2006 部分度量元示例

度量名称	度 量 目 的	应用的方法	测量、公式及数据元计算	测量值解释
功能的充分性	被评价的功能的充分程度如何	适于执行特定任务的功能数与评价的功能数相比较	$X=1-A/B$ A=在评价中检测出有问题的功能数 B=被评价的功能数	$0.0\leqslant X\leqslant 1.0$ 越接近1.0,越充分
响应时间	完成一项规定任务所花费的时间? 系统响应一项规定的操作之前要花去多长时间	开始一项规定的任务。 测量为完成其操作所花费的时间。 保持每次尝试操作的记录	T=(获得结果的时间)-(完成命令输入的时间)	$0<T$ 越快越好
吞吐量	有多少个任务能在给定的时间周期内成功地执行	测量完成任务而进行的操作所花费的时间; 保留每次尝试操作的记录	$X=A/T$ A=完成的任务个数 T=观察的时间段	$0<X$ 越大越好
估计潜在的故障密度	将来可能出现的故障问题有多少	对在一定的试验周期内检测到的故障数进行计数,并用可靠性增长估计模型来预测未来潜在的故障数	$X=\{ABS(A1-A2)\}/B$ (X:估计残存的潜在故障密度) ABS()=绝对值 $A1$=在软件产品中预测的潜在故障总数 $A2$=实际已检测到的故障总数 B=产品的规模	$0\leqslant X$ 取决于测试阶段,在以后的阶段中X值越小越好

1.4.3　ISO/IEC 25010—2014 软件质量模型

软件质量和软件需求分析、软件质量评价都存在密切的关系,但是这些技术要求都定义在不同的标准中,使得用户在综合使用标准过程中存在很大的困难。同时,标准之间也存在不一致或者不协调的方面。在意识到这些问题后,国际标准化组织ISO和IEC的联合技术委员会软件工程分技术委员会,ISO/IEC JTC1/SC7整合相关的标准,提出了系统与软件质量要求和评价SQuaRE(Systems and software Quality Requirements and Evaluation)系列标准,并以25000作为这些标准的统一编号。SQuaRE系列标准如图1-7所示,包括质量管理分部、质量模型分部、质量要求分部、质量评价分部以及扩展分部。

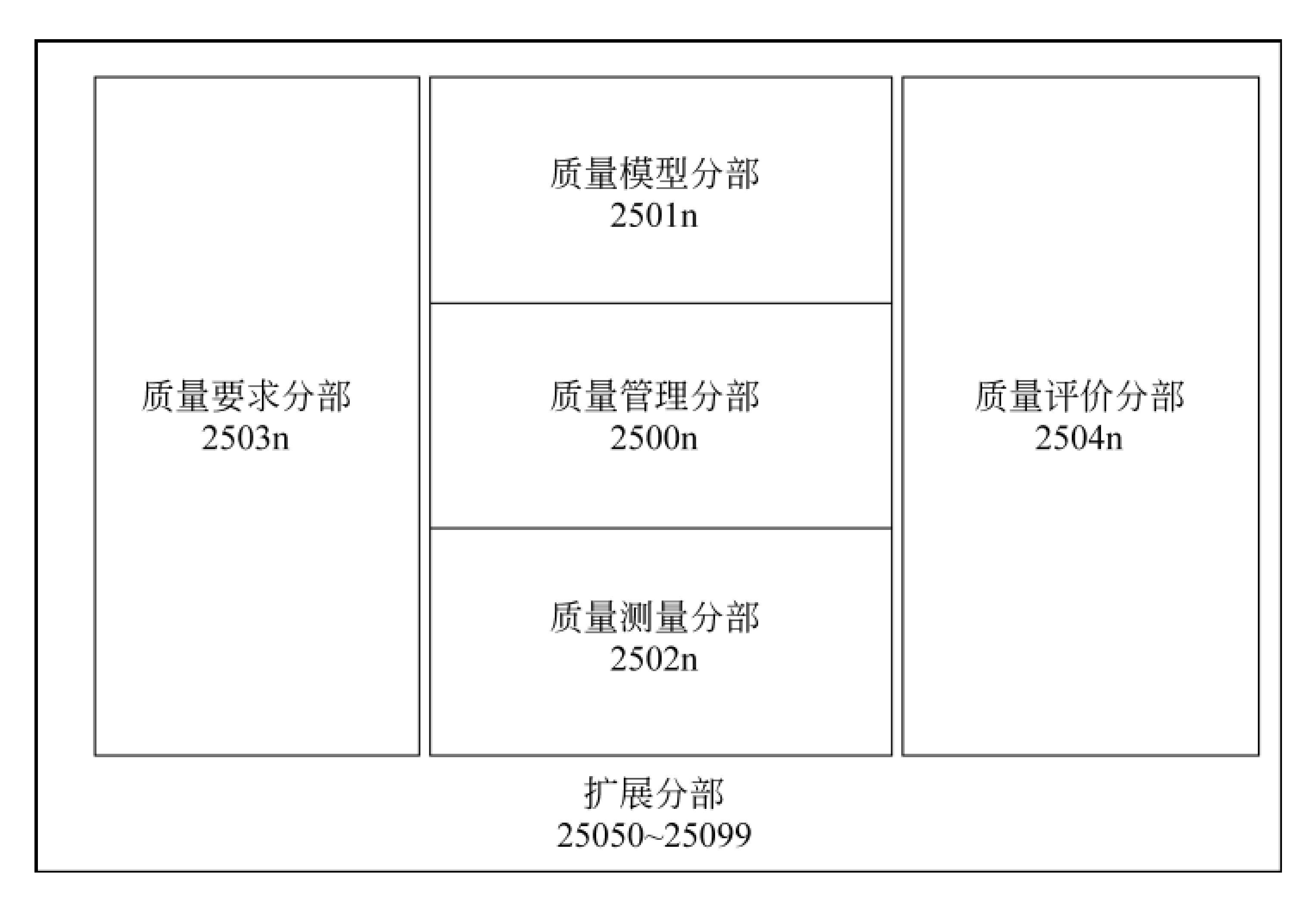


图 1-7　系统与软件质量要求和评价 SQuaRE 系列标准框架

质量管理分部定义了 SQuaRE 系列标准的全部公共模型、术语和定义。

质量模型分部给出了产品质量模型、使用质量模型和数据质量模型。

质量测量分部包括软件产品质量测量参考模型、质量测量的数学定义及其应用的实用指南。给出了软件内部质量、软件外部质量和使用质量测量的示例。定义并给出了构成后续测量基础的质量测量元素(QME)。

质量要求分部在质量模型和质量测量的基础上规定质量要求。这些质量要求可用在要开发的软件产品的质量需求抽取过程中或用作评价过程的输入。

质量评价分部给出了软件产品评价的要求、建议和指南,并给出了作为评价模块的测量编制支持。

SQuaRE 的扩展分部目前包括就绪即用软件的质量要求,易用性测试报告行业通用格式等。

ISO/IEC 25010 质量特性规定了软件产品质量模型,由 8 个质量特性组成,如图 1-8 所示。和 ISO/IEC 25010 对应的国家标准正在编制过程中,预计 2016 年正式发布。

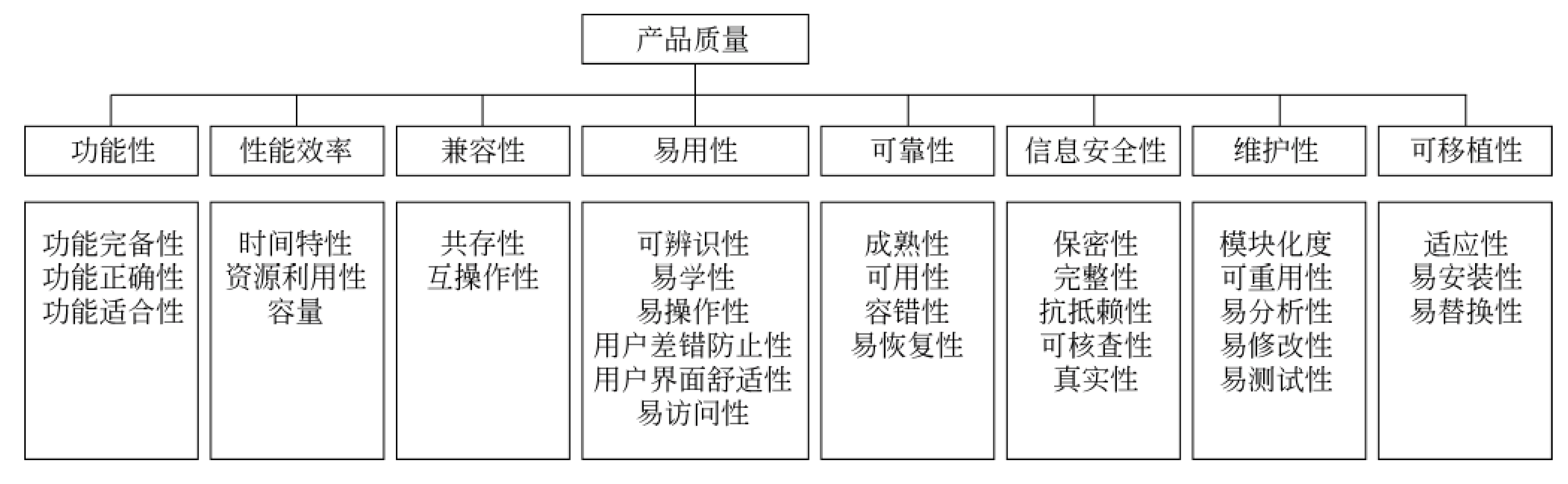


图 1-8　ISO/IEC 25010 产品质量模型

ISO/IEC 25010 定义了使用质量的 5 个特性：有效性、效率、满意度、抗风险性和周境覆盖度。其中，满意度、抗风险性和周境覆盖度进一步细分为若干子特性，如图 1-9 所示。

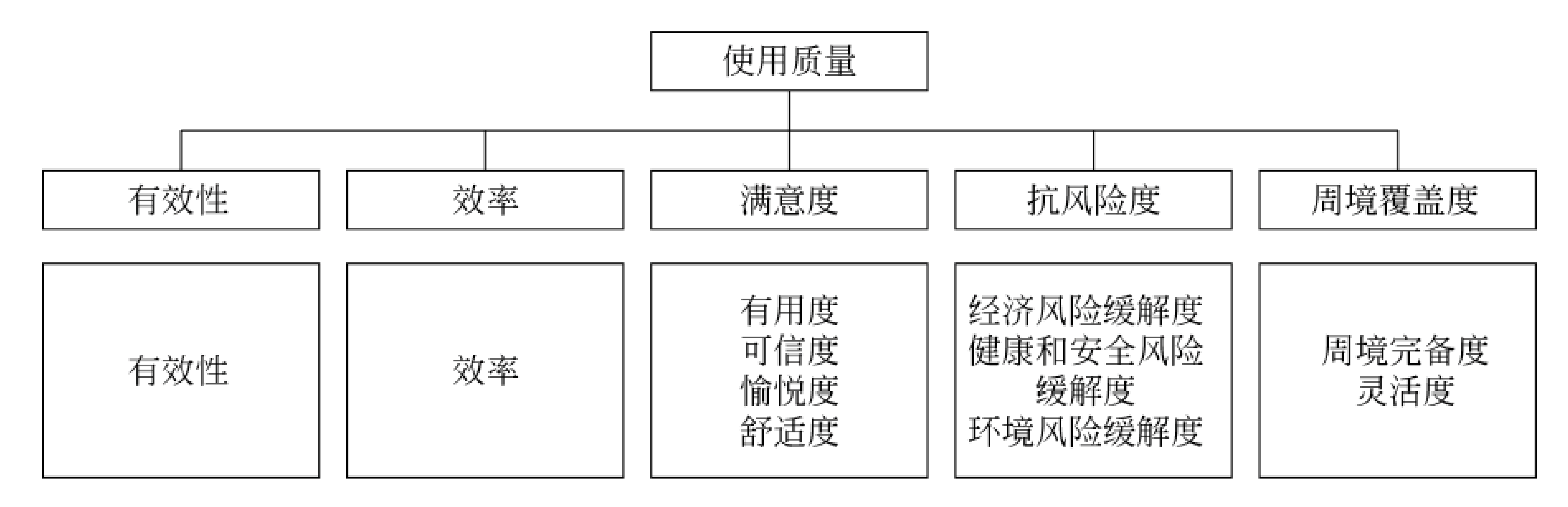


图 1-9　ISO/IEC 25010 使用质量模型

ISO/IEC 25010 仅定义了产品质量和使用质量的模型，而每一个子特性中的计算方法分别在 ISO/IEC 25022、ISO/IEC 25023 中定义，这两个标准在 ISO/IEC 处于 DIS (Draft International Standard)阶段。总体质量和度量元之间的关系，如图 1-10 所示。

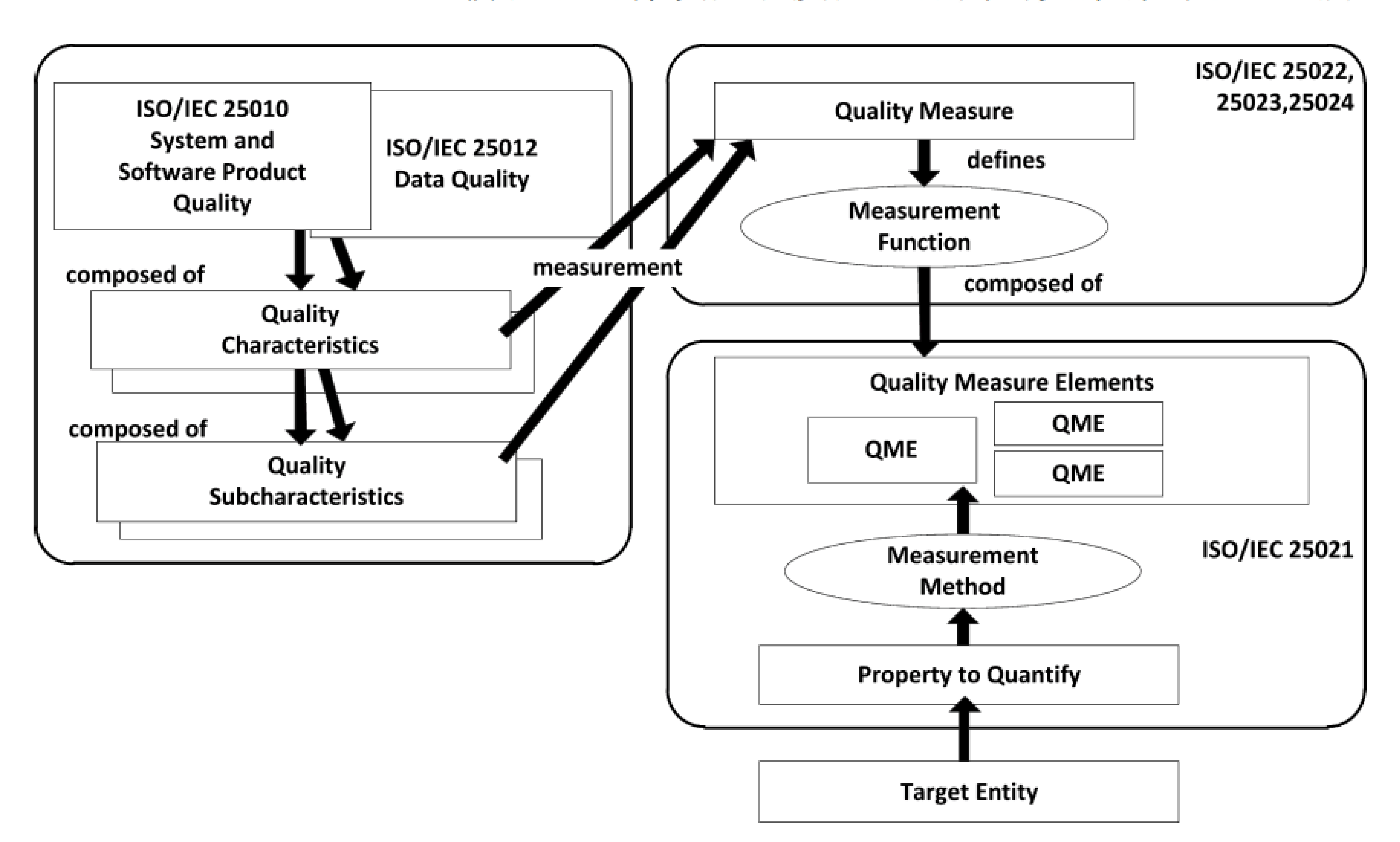


图 1-10　软件质量和度量元之间的关系

1.4.4　基于质量模型的软件测试标准

GB/T 25000.51—2010《软件工程　软件产品质量要求和评价(SQuaRE)商业现货(COTS)软件产品的质量要求和测试细则》依据 GB/T 16260.1—2006《软件工程产品质量 第 1 部分 质量模型》标准定义的软件质量模型提出了质量要求的说明和测试细则。从产品说明、用户文档集、软件质量三个方面提出要求。随着云计算以及 Web 服务的不断流行，软件由产品逐渐向服务演化，以出售产品许可证(License)方面的软件正在日益减

少，大量软件向信息服务的形式转变。这些软件的评测需求和商业现货软件存在很大的不同。笔者向国际标准化组织建议，修订 ISO/IEC 25051：2006 的适用范围，并得到了国际标准化组织的认可。在 2014 年发布的 ISO/IEC 25051：2014《软件工程　系统与软件产品质量要求和评价（SQuaRE）就绪即用软件产品（RUSP）的质量要求和测试细则》中重新做了修订，将对象由商业现货软件 COTS 调整为就绪可用软件产品 RUSP，同时对应的软件质量模型调整为 ISO/IEC 25010：2011。

1.5　软件测试模型

目前，软件测试已经发展成为软件质量保障的重要组成部分。不同的软件测试和软件开发交互而形成的过程模型，对软件开发的过程保障具有不同的影响，从而形成不同的软件测试模型。常见的软件测试模型包括：V 模型、W 模型和 H 模型。

1.5.1　V 模型

“瀑布模型”是软件工程领域最著名的开发模型之一，一直被业界所广泛认可和采用。瀑布模型将软件开发周期划分为制定计划、需求分析、软件设计、编码实现、软件测试和运行维护等 6 个基本活动，具有自上而下、相互衔接的固定次序。瀑布模型和传统的工程管理非常类似，在生存周期的不同环节，存在非常严格的先后次序关系，后一个环节基于前一个环节已经完成的前提执行。在 V 模型中，左边是需求分析、概要设计、详细设计以及编码实现，与其对应的分别是验收测试、系统测试、集成测试和单元测试。其形状和字母 V 非常类似，故称为 V 模型。图 1-11 给出了 V 模型的示意图。

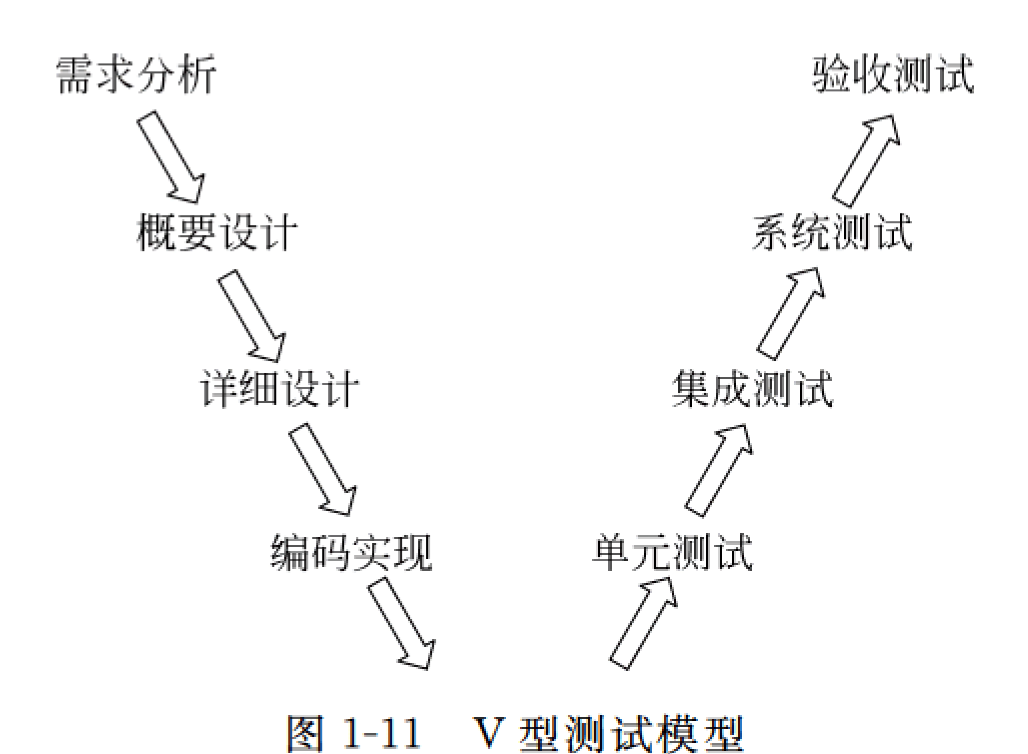


图 1-11　V 型测试模型

在需求分析、软件设计、编码实现三个环节产生了不同的系统输出，而不同层次的输出可以推导出不同层次的测试。例如，在需求分析阶段，从用户业务需求的层次上描述了系统，产生的需求规格说明书和用户的需求是最接近的，而依据需求规格说明书可以设计验收测试。在编码阶段，产生的是程序代码，依据代码可以设计单元测试用例。V 模型明确地阐述了测试过程中存在的不同级别，描述了测试阶段和开发过程期间各阶段的对应关系。

(1) 单元测试从代码实现逻辑上测试其是否正确;

(2) 集成测试应根据详细设计检测不同程序模块之间的接口是否满足软件设计的要求;

(3) 系统测试应检测系统功能、性能的质量特性是否达到系统要求的指标;

(4) 验收测试确定软件的实现是否满足用户需要或合同的要求。

在V模型中,把测试作为编码之后的最后一个活动,需求分析等前期产生的错误直到后期的验收测试才能发现,这是V模型较为明显的不足之处。

1.5.2 W模型

根据软件测试经济学,软件缺陷发现的越晚,其修复的成本也就越高。在V模型中,需求分析阶段所引入的错误,将会在验收测试才能发现。在实际工程实践中,需求错误所产生的损失远比代码错误所造成的损失大。Ron Patton在《软件测试》中指出:如果修复在需求分析阶段发现一个缺陷的成本为1$,那么如果在编码或者测试阶段发现同一个缺陷,其修复的成本大约在10~100$之间。如果是客户发现这个同样的缺陷,那么修复成本可能高达上千美元。V模型无法体现"尽早地和持续测试"的原则。W模型在V模型中各个开发阶段,均增加同步的测试,以实现尽早测试。W模型由Evolutif公司提出,相对于V模型,W模型能够使得开发和测试并行执行。图1-12给出了W模型的示意图。

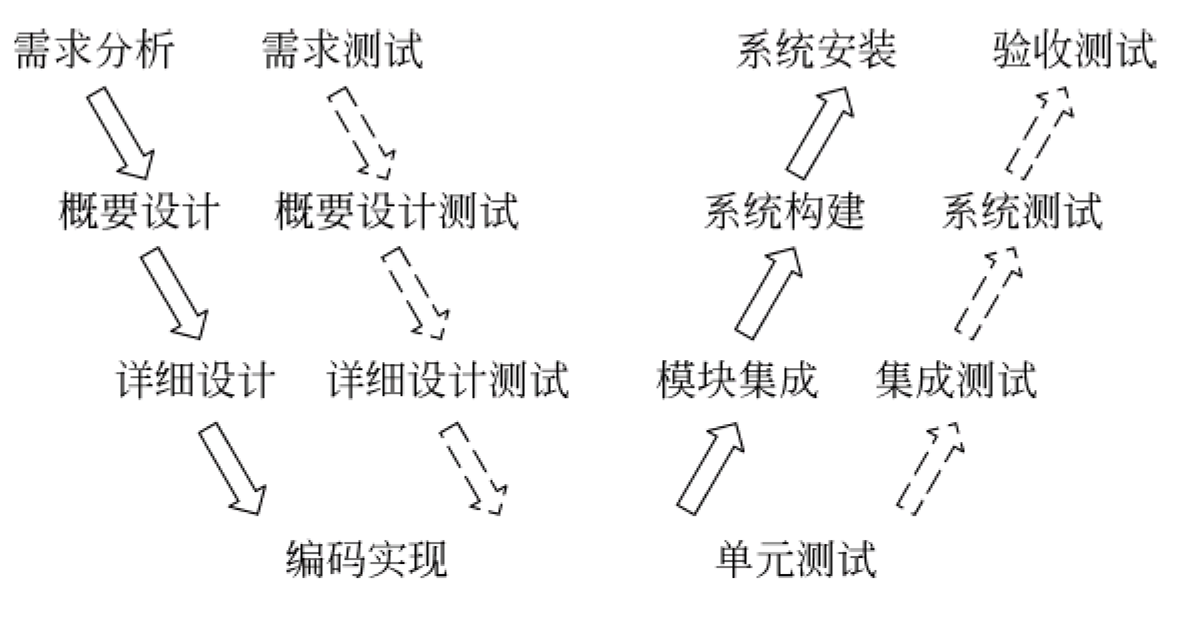


图1-12 W型测试模型

W模型实际上由两个V模型所构成,其中一个开发V模型,另一个测试V模型。在开发V模型中,涵盖了需求分析、概要设计、详细设计、编码实现,模块集成、系统构建和系统安装。在W模型中测试是一个广义的概念,例如,需求分析相对应的需求测试实际上是需求评审。在V模型中,需求测试、概要设计测试和详细设计测试遵循IEEE《软件验证与确认(V&V)》规定的原则。

虽然W模型对于V模型做了改进,W模型还是划分了较为严格的前后次序关系,例如一定是在完成所有软件单元测试的基础上,执行软件集成测试。

1.5.3 X 模型

随着软件规模的不断增大，在 W 模型中需要等到所有的单元编码完毕以后，才能执行系统集成。如果在集成过程中，发现一个单元存在软件缺陷，所有的集成无法继续执行，导致大量的等待时间。X 模型的思路是，在几个程序片段完成开发和测试以后，立即开始系统集成以及测试，如有更多的程序片段，也以此类推执行。这样，可以大大增加不同程序片段之间的并行度。某一个程序片段发现的软件缺陷，其影响的范围也仅仅是本次参加集成的片段。图 1-13 描述了一个 X 测试模型的原理。

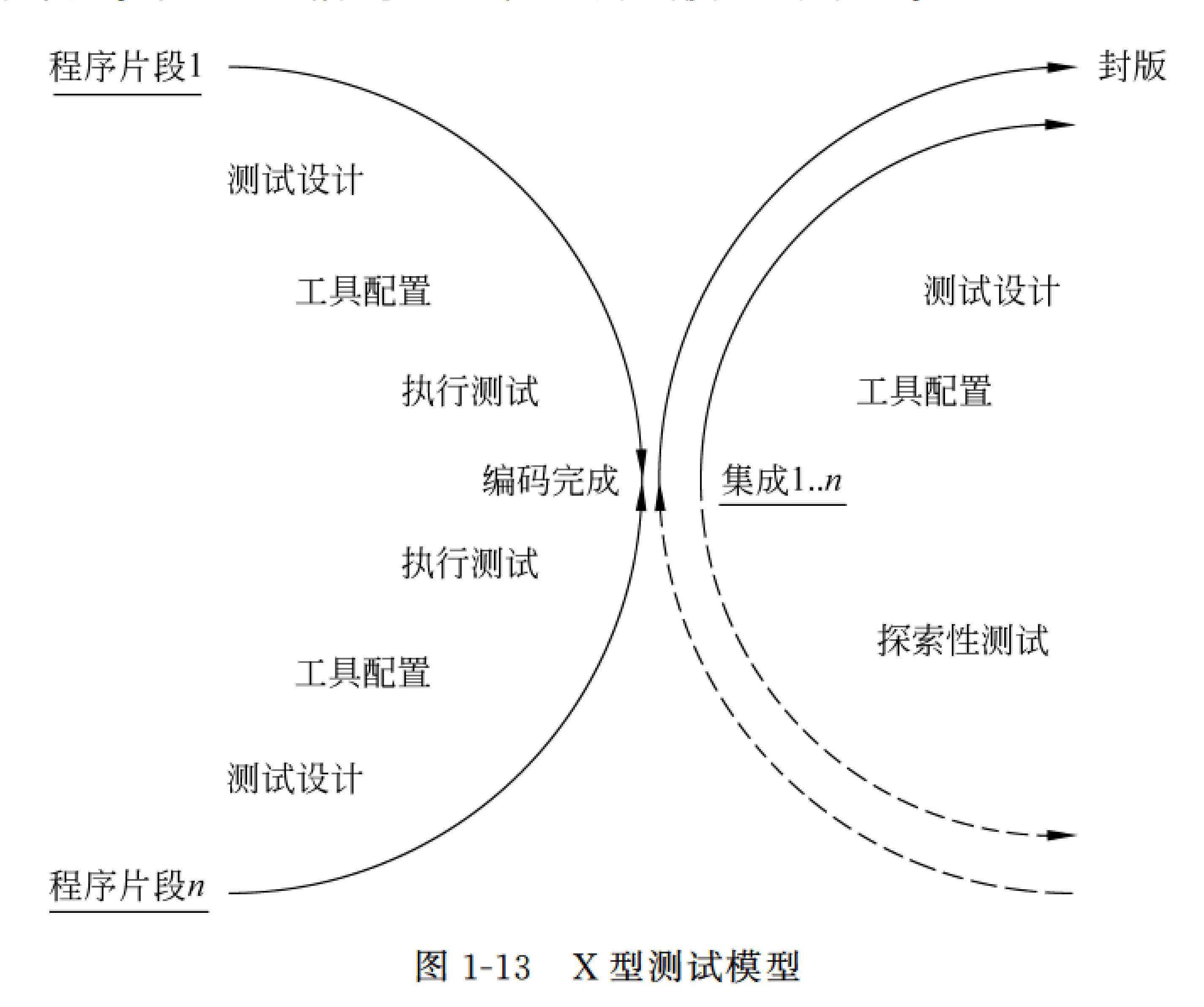


图 1-13 X 型测试模型

在 X 模型中，左边描述了两个独立的程序片段的开发和编码过程。每一个程序片段包括测试设计、工具配置、测试执行等阶段。在两个独立程序片段完成以后，执行已经完成程序片段的集成。如果所有的片段都完成执行，那么整个程序可以正式发布，如右边的上边所示。这个过程需要反复多次，在中间的集成节点上，集成次数存在多次，在 X 的交叉点上的 1..n 表示集成多次。在不同的集成过程中，可以安排探索性测试。

1.5.4 H 模型

由于软件测试本身的工程化程度不断增加，软件测试包括测试计划、测试准备、测试设计、测试执行、测试分析等阶段。在 H 模型中，软件测试是一个独立的流程，贯穿产品整个生命周期，与其他流程并发地进行，如图 1-14 所示。

软件测试作为一个相对独立的流程，只要测试条件就绪，并且测试准备活动已经完成，测试执行活动就可以执行。模型中所描述的其他流程，可以是开发流程，也可以是软件质量保障 SQA 流程，或者是其他可以触发软件测试的流程。H 模型同时考虑系统效

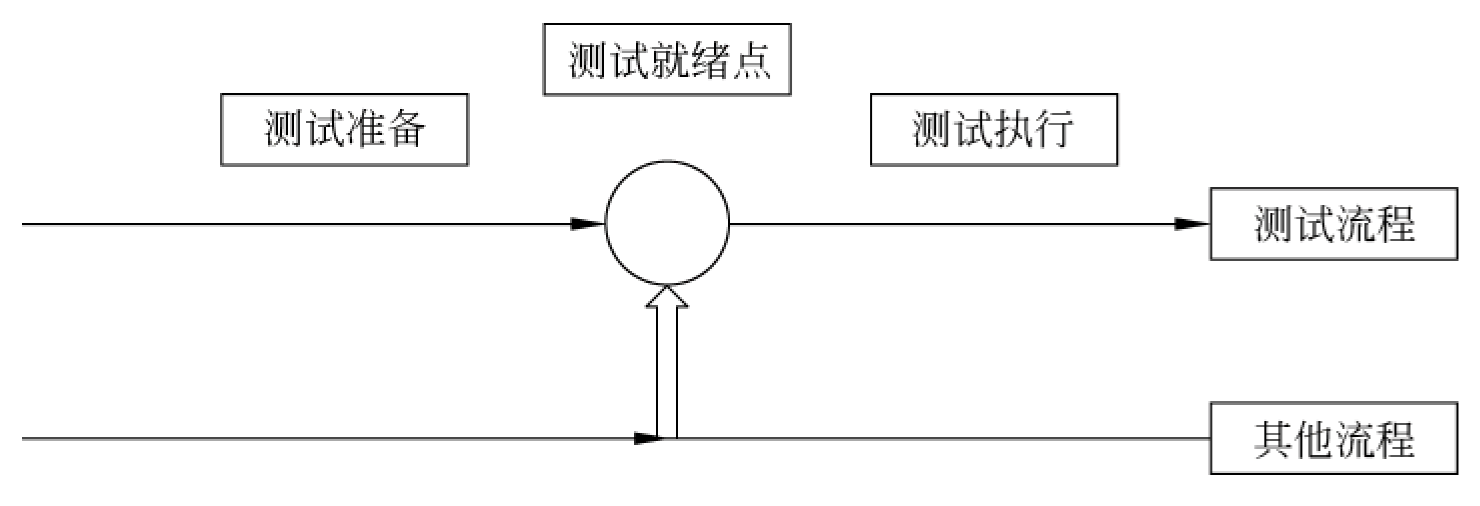


图 1-14　H 型软件测试模型

率和灵活性，被应用到各种规模、各种类型的软件项目上。不同的测试活动可以是按照某个次序先后进行的，但也可能是反复的，只要某个测试达到准备就绪点，测试执行活动就可以开展。

1.6　软件测试的局限性

1.6.1　软件测试的覆盖问题

软件开发过程，实际上是程序代码实现规格说明书的一个过程。软件质量保障的目的之一就是尽可能地使程序的实现和规格说明书保持一致。假设利用 S 表示规格说明书(Specification)，而 P 表示程序实现(Program)。在实际软件项目中，P 和 S 之间并不能完全做到一致，如图 1-15 所示的维恩图(Venn Diagram)清晰地表明了 P 和 S 之间的关系。

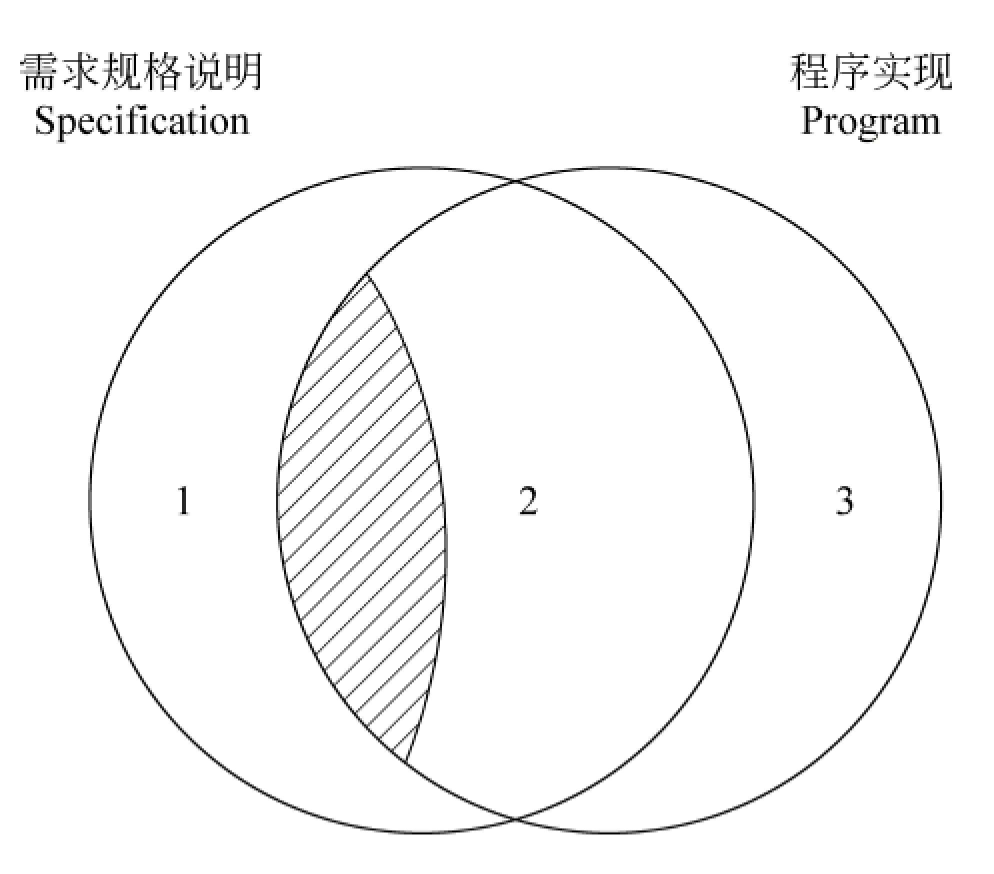


图 1-15　程序实现 P 和规格说明 S 之间的关系

在软件生存周期的不同环节，信息的沟通和传递都存在变形，最常见的形式是信息丢失、信息扭曲以及信息增加。例如，在需求分析阶段，需求规格说明书表达的是 ABBD，而经过多层的技术人员的传递，最后到开发人员可能存在以下几种情况。

(1) 信息遗漏：遗漏中间的一个 B，认为其得到的信息是 ABD。

(2) 信息扭曲：将中间的一个 B 误听成为 D，认为其得到的信息是 ABDD。

(3) 信息增加：多听了一个 B，认为其得到的信息是 ABBBD。

图 1-15 表示了规格说明和程序实现之间的关系。左边圆圈表示了规格说明 S 描述的内容，在这个圈中标注为 1 的部分，也就是最左边的月亮型部分，在传递过程中被丢失，程序 P 并没有实现 S 规定的内容。在维恩图中，中间两个图的交集表示程序实现了规格说明的部分，其中右边空白部分（标注为 2 的部分）表示正确实现，而左边阴影部分表示程序实现了但是存在错误，其情况类似信息扭曲。而右边的月亮型部分，标注为 3 的部分表示规格说明书没有规定，但是程序实现了，类似信息增加的情况。

若根据需求规格说明书来设计测试用例，那么称为黑盒测试，若根据程序实现来设计测试用例，那么称为白盒测试。但是无论是白盒测试还是黑盒测试，都无法恰好完全覆盖需求。需求规格说明、程序实现、软件测试三者之间的关系如图 1-16 所示，其中左边圆部分表示需求规格说明书，右边圆部分表示程序实现，下面圆部分表示测试用例。三个圆将空间分割成三个不同的部分，分别用 1～9 标注。

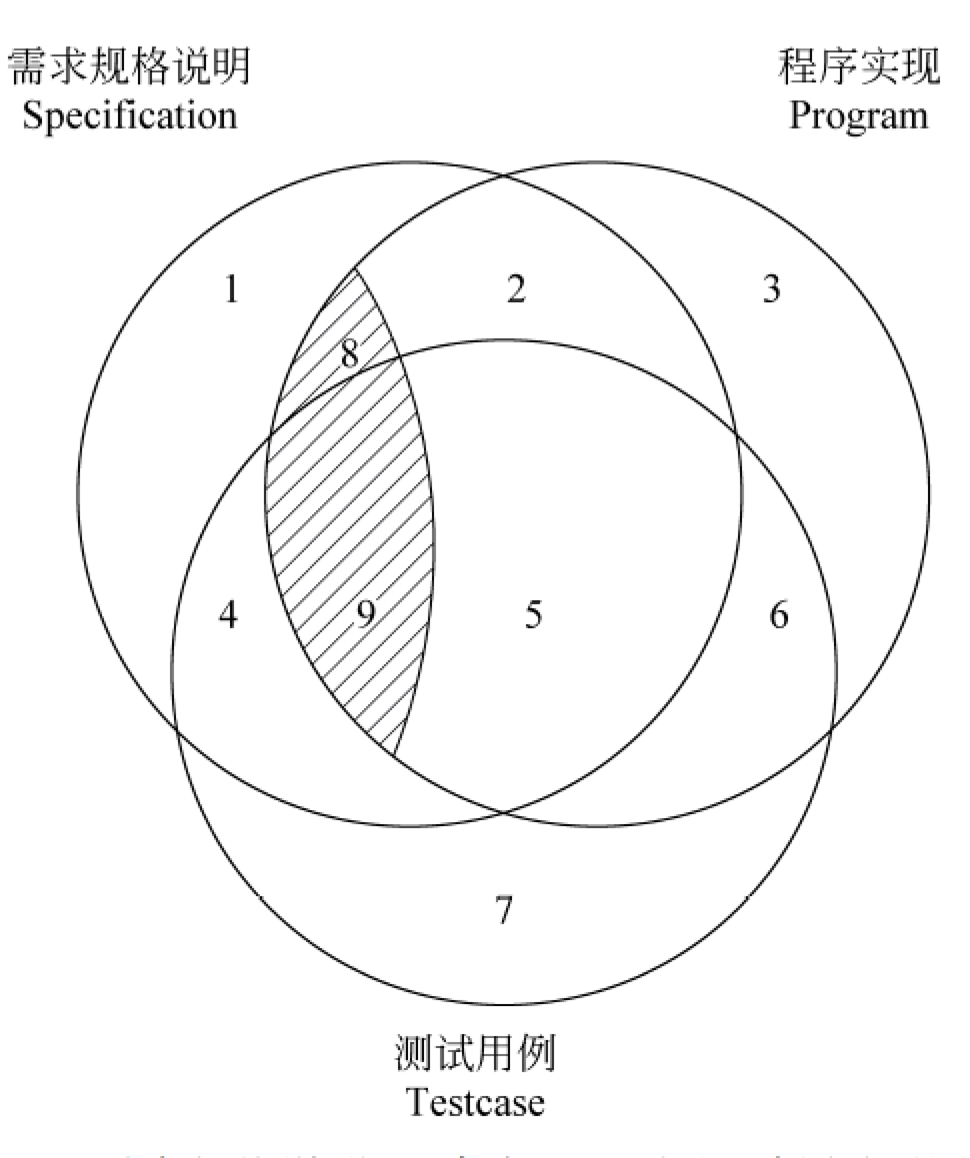


图 1-16　需求规格说明、程序实现和测试用例之间的关系

其中，区域 1 表示需求规格书中定义的，但是程序没有实现，测试用例也未覆盖到该区域。标记为 2 的部分，是需求规格说明定义了，程序已经实现，但是测试用例并未覆盖该区域。标记为 5 的区域，是程序实现规格说明的内容，测试用例也覆盖规格说明的内容，这个区域如果越大，说明该企业的软件工程管理越到位。标记为 3 的区域，表示程序实现超出需求规格说明未说明的部分，测试用例也未覆盖这部分。标记为 6 的区域，表示程序实现超出了规格未说明的部分，但是测试用例覆盖该部分的程序实现。标记为 7 的区域，表示测试人员设计测试用例，原以为能够覆盖需求或者程序实现，但是由于理解上的偏差或者技术能力上的限制，设计的测试用例未能覆盖需求或者程序实现。图中带阴影的部分，是程序实现覆盖需求，但是没有正确实现的部分。标记为 8 的部分，表明程序中仍然存在没被测试用例覆盖的带有缺陷的部分。标记为 9 的部分，表示测试用例覆盖了程序中存在的缺陷部分。

一个理想的状态是三个圆完全重叠，程序实现恰好完全覆盖了需求，同样测试用例也恰好覆盖了需求，但是实际上要达到这种状态是非常困难的。和信息在传递过程中的变形一样，要完全发现并消除缺陷在实际上是不可行的。

1.6.2 穷举测试的局限性

Robert V. Binder 在《面向对象的系统的软件测试》中指出，穷举测试在现实上是不可能的，无论是黑盒测试还是白盒测试都存在一定局限性：黑盒测试存在状态爆炸问题，白盒测试存在路径爆炸问题。同时，缺陷的隐蔽性也给测试带来一定的困难。

1. 黑盒测试存在状态爆炸

黑盒测试无须知道程序实现的内部结构，可根据其输入和输出的对应关系设计测试用例。软件的输入和输出的组合关系构成软件的状态，状态数量随着输入输出参数个数的增加呈指数级增长，其产生的状态数量使得所需要的计算时间超出实用的范围。从黑盒角度看，一个程序 P 可看成将有 n 个输入参数的实体转换为 m 个输出参数的实体，如图 1-17 所示。在这个模型中，假设 n 个输入参数的取值分别为 $I_1, I_2, \cdots, I_n$，如果单独考虑输入参数的组合关系，那么输入总数为：

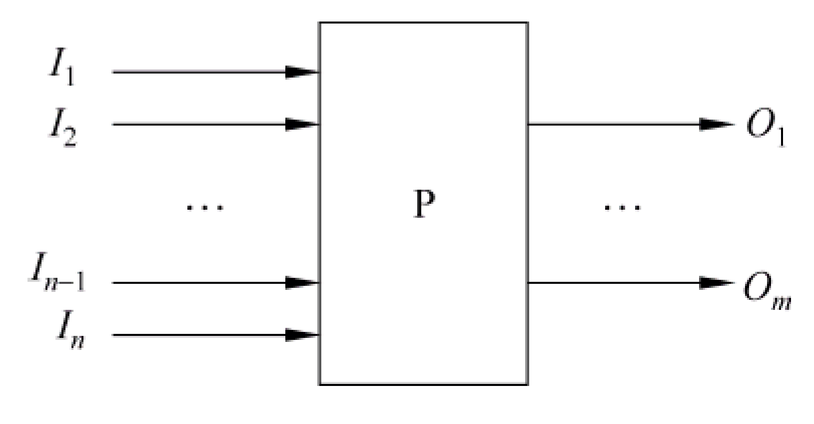


图 1-17 黑盒测试模型

$$I_1 \times I_2 \times I_3 \times \cdots \times I_n$$

例如，在同一个平面上，需要判断三条线段是否能够构成一个三角形。每一条线段都存在两个端点，每个端点的坐标分别为 $P(x, y)$，那么程序共需要 6 个坐标作为输入，每个坐标包含 x 轴和 y 轴，取值范围为 1～10 之间的整数，三条直线共有 10^{12} 条直线的组合。假设每秒钟测试 1000 条直线，大概需要 317 年时间。

在实际的测试工作中，从成本收益的角度看，没有必要测试每一个输入。执行组合测试一般需要先使用其他测试方法，产生能够代表不同参数输入发现缺陷能力的部分数据，然后产生组合。而相同的发现缺陷能力在事先是无法证明的，只是一种理论上的分析或者假设。

2. 白盒测试存在路径爆炸

软件程序的结果包括顺序结构、分支结构、循环结构，并且这些结构存在网状的嵌套形式，其产生的路径数量也极其巨大。例如，在循环内部包含分支就是一种比较常见的程序结构。如果分支数为 m，循环数为 n，如图 1-18 所示。那么当循环变量为 1 时，共有 m 条路径。循环变量为 2 时，其自身的路径也为 m 条。考虑循环变量为 1 时和循环变量为 2 时的路径交叉，依据乘法原理，两次循环的路径为 $m \times m = m^2$，当循环变量达到 n 时，总的测试路径数达到 m^n。显然路径随着循环次数呈指数级增长。关于路径的测试方法，将在控制流测试一章中进行展开讨论。

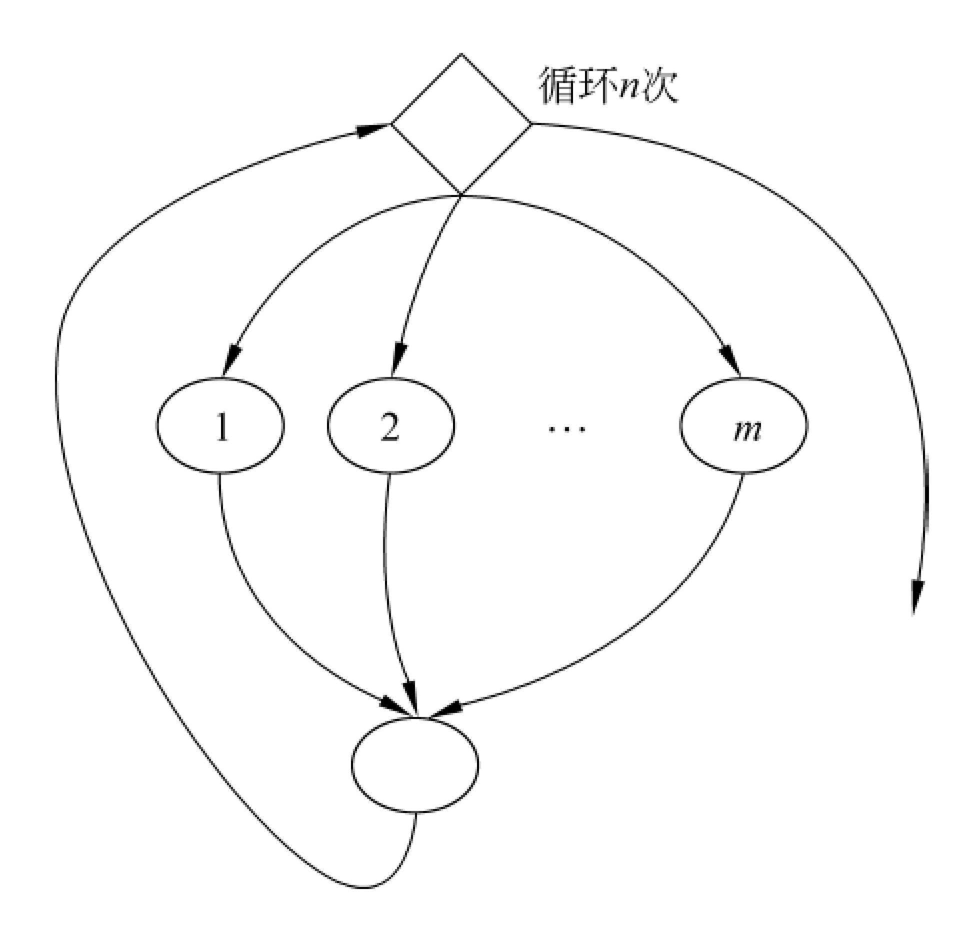


图 1-18　白盒测试路径示意图

1.6.3　缺陷的隐蔽性

在执行软件测试时,人们总是希望当执行到具有错误缺陷的代码时,就会引发故障,从而发现该缺陷的存在。但是实际情况并没有如此理想,有时即使执行到了错误的代码,也由于种种原因或者巧合,导致结果和正确的代码一致,从而无法发现该错误,这就是缺陷的隐蔽性。

下面以顺序统计量为例子说明这个现象。顺序统计量的一个特例是中位数,中位数是指它所在集合的"中间元素",当 n 为奇数时,中位数是唯一的,出现位置为 $n/2$;当 n 为偶数时,存在两个中位数,位置分别为 $n/2$(上中位数)和 $n/2+1$(下中位数)。更加一般的问题是顺序统计量:在一个由 n 个元素组成的集合中,第 i 个顺序统计量是该集合中第 i 小的元素。例如,最小值是第 1 个顺序统计量,最大值是第 n 个顺序统计量。在线性时间 $O(n)$内,在集合 S 中选择第 i 小的元素。

下面给出一个具体的实例。

输入:一个包含 n 个(不同的)数的集合 A 和一个数 i,$1\leqslant i\leqslant n$。

其中:$A=[3,7,10,5,2,8,1,1,15,16,17,14,18,19,11,10,1,1,19,19]$。

输出:元素 $x\in A$,它恰大于或者等于 A 中其他的 $i-1$ 个元素。

根据需求,程序 1-1 给出了一个具体的实现代码。在这个代码中,主要的实现函数是 selection,selection 函数利用类似二分查找的方法实现顺序统计量。给定一个基准,将数列划分为两个部分。在基准前面的数都小于基准,在基准后面的数据都大于等于该基准。基准可以随机选择,也可以利用其他的启发式方法进行设置。在程序中直接采用第 1 个元素作为基准。

程序 1-1　包含一个错误的线性时间求顺序统计量

```
#mink.py
#在线性时间内求第 k小的值
```

```
array= [3,7,10,5,2,8,1,1,15,16,17,14,18,19,11,10,1,1,19,19]

def swap(array,i,j):
    t= array[i]
    array[i]= array[j]
    array[j]= t
    return 0

def partition(array,low,high):
    x= array[low]
    m= low
    for i in range(low+ 1,high+ 1):
        if(array[i]< x):
            ++m    #m= m+ 1
            swap(array,m,i)
    swap(array,low,m)
    return m

def selection(array,left,right,k):
    if (left= = right):
        return array[left]

    if (left> right or k- 1> right- left):
        return- 1

    mid= partition(array,left,right)
    len= mid- left
    if(k- 1= = len):
        return array[mid]
    elif k- 1< len:
        return selection(array,left,mid- 1,k)
    else:
        return selection(array,mid+ 1,right,k- len- 1)

for i in range(1,len(array)+ 1):
    print selection(array1,0,len(array)- 1,i) ,' ',
```

为了说明缺陷的隐蔽性以及巧合性,将程序中所有的顺序统计利用循环语句输出:

1 1 1 1 2 3 8 7 8 10 10 11 14 15 16 17 18 19 19 19

而实际的结果应该为:

1 1 1 1 2 3 5 7 8 10 10 11 14 15 16 17 18 19 19 19

对于这个特定的数列而言，该程序实现除了在3和7之间一个位置出错以外，其他全部正确。

引起这个故障的原因比较隐蔽。在C、C++以及Java语言中，都存在自增的运算符++，而在Python语言中并不存在自增运算符，++m语句在Python语言中代表执行两次符号运算。对于一个正整数m而言，++m并不会引起数值m的任何变化。正确的语句应该是m=m+1。但是这个错误，针对这个特定的输入而言，其隐蔽性非常好，在20个顺序统计量中，仅有一个顺序统计量不正确。

1.6.4 软件测试的杀虫剂效应

Boris Beizer在1990年提出了软件测试的杀虫剂效应。研究发现，如果农民总是喷洒同一种农药，害虫会对这种农药产生一定免疫力，农药效力相比初期，其效果会大大下降，即所谓"杀虫剂"效应。在软件测试用例领域，用于描述软件多次被同一个方法测试，被测软件会对该测试方法产生免疫力的现象。

在各种构件化开发过程，中前期发现的各种缺陷模式，都通过验证或者校验功能集成在构件中，已经成为构件的必然属性。这些构件无须开发人员单独编写代码就对已有的测试方法具有天然的免疫能力。图1-19给出了一个软件测试杀虫剂效应的示意图。在第一阶段，软件开发人员大多将精力集中在业务功能的实现上，而导致了软件中存在很多缺陷。通过测试以后，发现软件中存在的缺陷。经过时间的不断积累，技术人员总结出了软件测试发现的常见缺陷，并分析其产生的原因，并形成能够自动防止这些缺陷发生的代码或者模块，形成一层自动的免疫壳。这些免疫壳，可以自动加载在业务功能外，而无须特定的编程。这样的代码提交给测试团队以后，那么测试团队在初期测试时所采用的方法，在具有免疫壳的业务功能上，都无法发现新的缺陷。

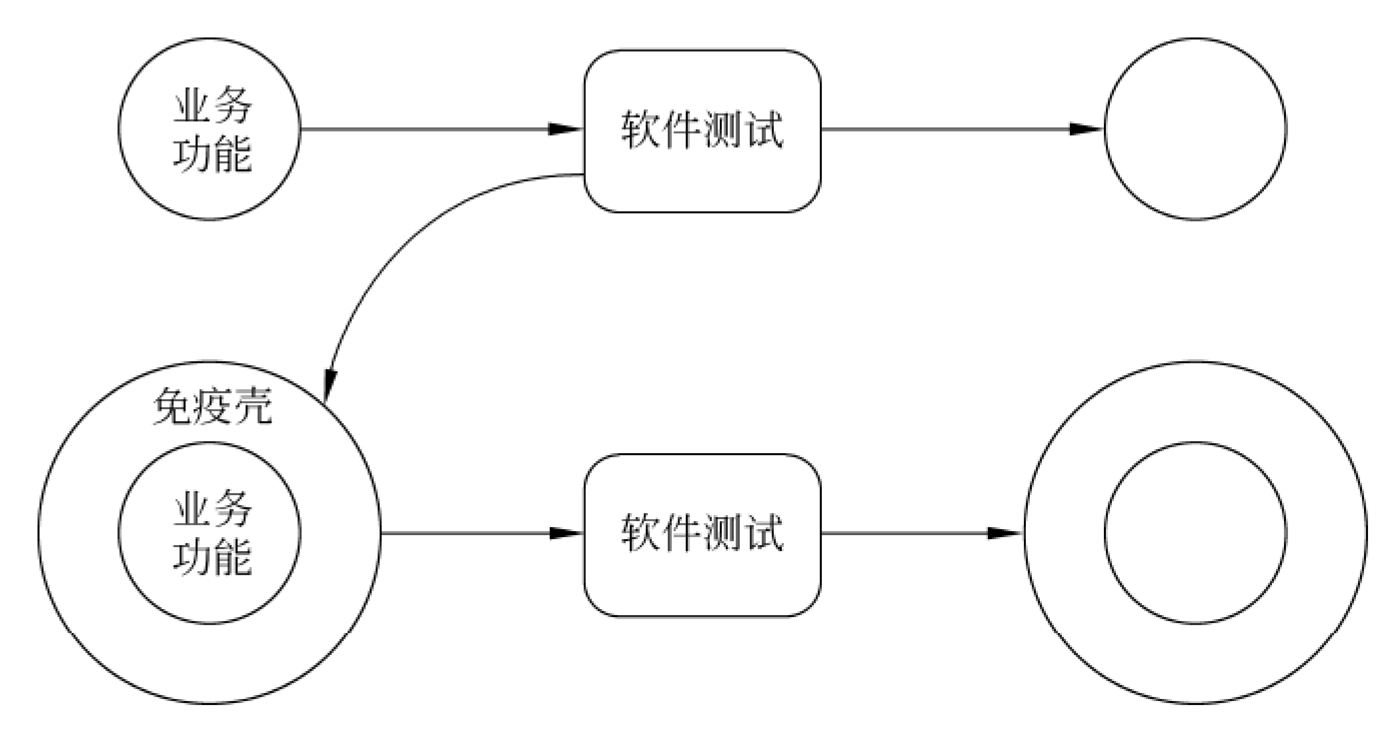


图1-19 软件测试的杀虫剂效应

以.NET中Web控件为例，提供了大量数据正确性校验的组件，包括：强制输入校验、比较校验、边界值校验、用户自定义校验等。RangeValidator控件检查用户的输入是否在指定的上下边界之间。可以检查数字、字母或日期对内的范围。可以将边界表示为常数。RegularExpressionValidator控件检查输入是否与正则表达式定义的模式匹配。

该验证类型允许检查可预知的字符序列，如社会保障号、电子邮件地址、电话号码、邮政编码等中的字符序列。图 1-20 给出一个电话号码的正则表达式校验，要求输入的格式必须是带括号的两位区号和两个 4 位号码组合，其中除括号以外所有的符号必须为数字。开发人员需要实现带有自动电话号码校验功能的模块，只要在控件窗口设置校验正则表达式即可，而无须编写专门的边界校验功能的代码。

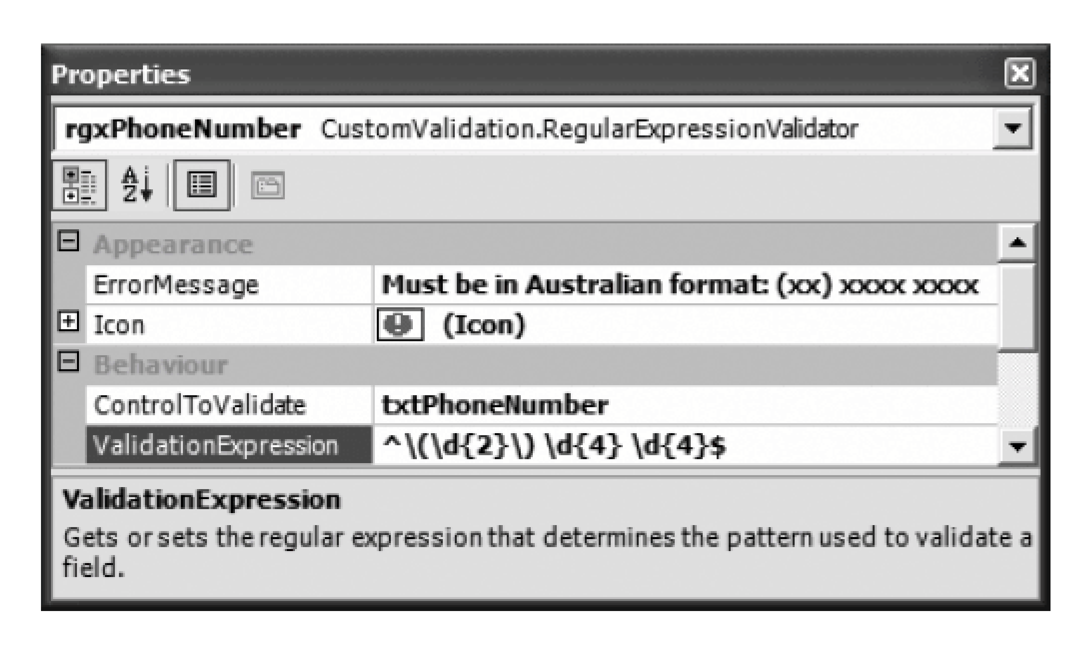


图 1-20 电话号码正则表达式校验

除了在前面所讨论的集成在控件内部的数据有效性校验外，随着面向方面的编程(Aspect Oriental Program，AOP)等新一代编程技术出现，在源码层次可以直接利用装饰器等技术实现对参数进行判断，防止调用出错，用于参数有效性检查的装饰器构成前面描述的免疫壳。程序 1-2 给出了一个关于免疫壳程序的示例，其中的业务功能是求斐波那契数列的值。显然，对于斐波那契数列，隐含的需求是输入参数必须是整数，并且值必须大于零，如果输入无效的参数将可能导致程序出错。而在考虑斐波那契数列的实现时，可以暂时不用考虑和程序健壮性相关的内容，而只需集中精力思考其核心算法思路。将这些额外的功能定义成为一个装饰器，而这个装饰器可以在不同的程序中复用，形成关于参数检查的免疫壳。

程序 1-2　具有参数有效性校验的程序

```
#paracheck1.py

def argument_test_natural_number(f):
    def helper(x):
            if type(x)==int and x>0:
            return f(x)
        else:
            #raise Exception("Argument is not an integer")
            return-1
    return helper

@ argument_test_natural_number
def factorial(n):
    if n==1:
```

```
        return 1
    else:
        return n * factorial(n-1)

for i in range(1,10):
    print(i, factorial(i))

print(factorial(-1))
```

软件测试的“杀虫剂”效应，使得软件测试的技术必须不断地升级，才能发现软件中存在缺陷。在测试初期，少量的测试用例将会发现较多的缺陷，在后期测试能够发现的缺陷数量将逐渐趋缓，甚至缺陷的数量在一段周期内停止增长。随着杀虫剂效应的显现，同样的测试用例能发现的缺陷数量在减少，测试的难度在不断增加。

1.7　软件测试的分类

1.7.1　软件功能测试分类

测试包括基于规格说明书的测试（黑盒测试）和基于控制流的测试（白盒测试），如图 1-21 所示。基于规格说明书的测试包括边界值、等价类、决策表、因果图、状态机、UML、Petri 网、Z 规约等。但是边界值、等价类、决策表、因果图等方法对于庞大的系统表达存在较大的困难，本书将其纳入到传统的黑盒测试模块。将状态机、UML、Petri 网、Z 规约等称为基于规格说明的测试方法。白盒测试方法包括语句覆盖、判定覆盖、条件覆盖、MC/DC 覆盖、基本路径覆盖等。除此以外，测试还包括冒烟测试、错误猜测测试、随机测试、故障树测试、基于蜕变的测试方法等。

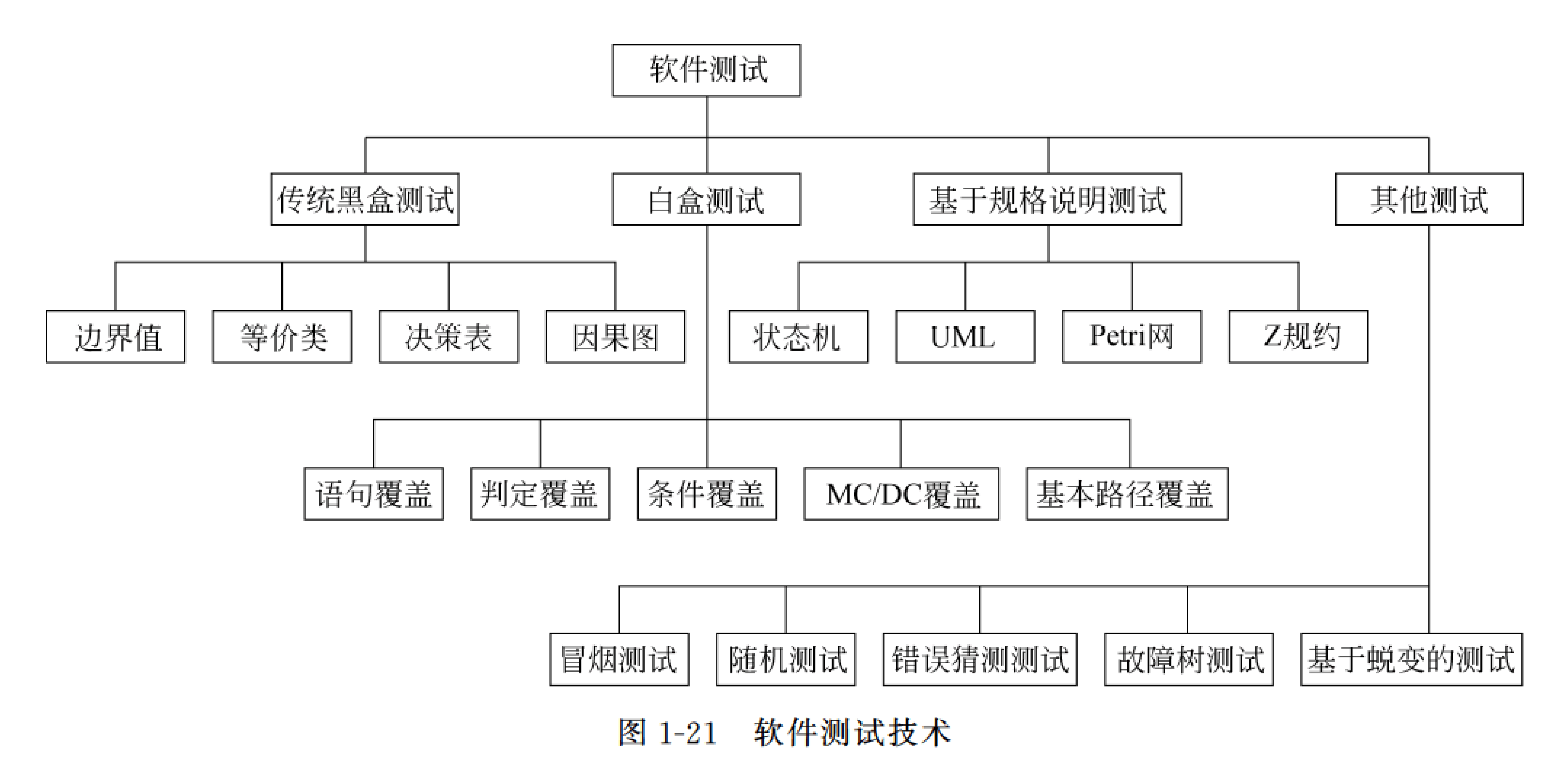


图 1-21　软件测试技术

1.7.2 根据测试阶段分类

在V模型中，和软件生存周期存在对应的测试阶段，如图1-22所示。软件包括项目型的软件和产品型的软件。如果是一个项目型软件，软件测试包括单元测试、继承测试、系统测试、验收测试。如果是一个产品型软件，软件测试包括单元测试、集成测试、α测试、β测试。

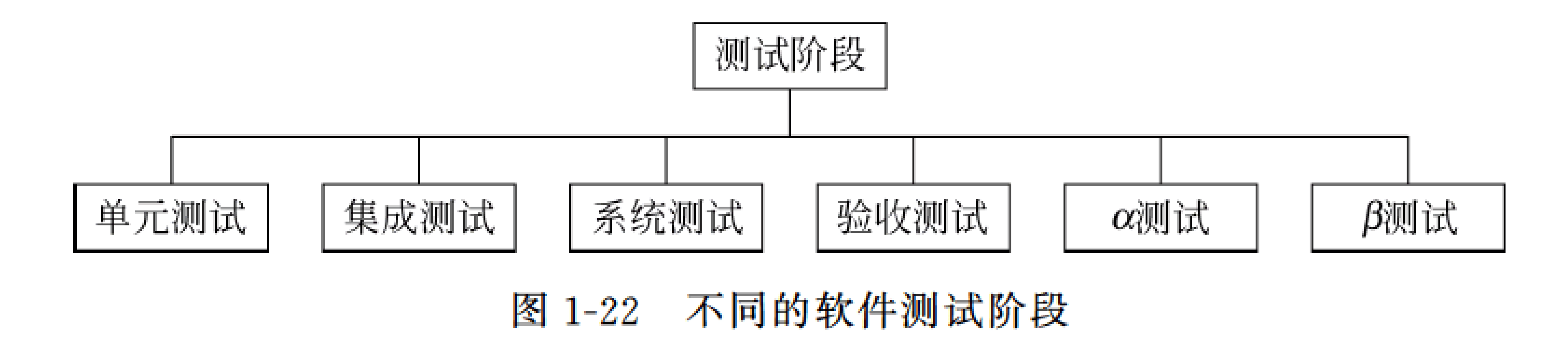


图1-22 不同的软件测试阶段

1. 单元测试

单元是规定的最小的被测功能模块，单元测试是指对软件中的最小可测单元进行检查和验证。单元可以是一个函数、一个类或者类的一个方法。一个单元通常具有下列基本属性。

(1) 明确的功能；

(2) 可定义的规格；

(3) 与其他单元接口的清晰划分。

单元测试的目的包括验证代码是与设计相符合的，发现在编码过程中引入的错误。在单元测试前，一般先评审代码是否符合规范，包括编码规范、命名规范等。单元测试动态运行代码，以检查运行实际结果的正确性。在执行单元测试前，先定义好白盒测试的基本要求，例如，所有设计的测试用例必须全部执行、语句覆盖率达到100%、分支覆盖达到90%等。除了覆盖率以外，还应关注：

(1) 参数的属性、顺序、个数是否正确，这个检查在支持不定参数传递时必须重点关注。

(2) 是否修改只作输入用的形参，否则可能导致数据的错误修改；如果只用于输入，一般采用传值的方式传递，否则可以采用传地址的方式。

(3) 约束条件是否通过形参来传送。检查对于约束条件的判断在传入单位以后进行检查，还是在传入之前进行检查。

在Python语言中，参数传递的对应关系非常灵活，支持位置传递、关键字传递、默认值传递、包裹传递、包裹关键字传递等。程序1-3给出了参数传递的对应关系的简单例子。函数func和func1演示位置传递、关键字传递、默认值传递使用的例子。在位置传递方式中，实参和形参的对应关系完全依赖位置。使用位置传递，相对于关键字传递发生错位概率较大。标注了参数名的就要按参数名传递，打乱顺序的情况下一定要加参数名，否则会混乱的。没有默认的实参情况下就会依次传递，如果不够，后面的会自动去取自己

的默认值。函数 func2 和 func3 是两种包裹关键字传递,带 * 表示所有的参数被 name 收集,根据位置合并成一个元组(tuple),而**表示 name 是一个字典,收集所有的关键字,传递给函数 func3。由于包裹关键字方式,对于参数的个数和形式都没有特定的约束,需要在单元内部执行参数个数和有效性检查。

程序 1-3　Python 参数传递对应关系例子

```
def func(a,b,c):
    return a+b+c

sum=func(1, 2, 3)                  #位置传递,
sum1=func(c=3,b=2,a=1)             #关键字传递
sum3=func(1,c=3,b=2)               #传递混合

def func1(a,b,c=10):
    return a+b+c
sum4=func1(3,2)
sum5=func1(3,2,1)
def func2(* name):
    sum=0
    for i in name:
        sum=sum+i
    return sum

sum6=func2(1,2,3,4,5)
print sum6

def func3(**name):
    sum=0
    for key in name:
        sum=name[key]+sum
    return sum

sum7=func3(a=1,b=2,c=3,d=4)
print sum7
```

在 Python 中所有的变量都是引用。利用变量传递的方式就是传值,若将一个值放在列表中,通过列表名进行传递,其在效果上等同于传地址。在程序 1-4 中,左边表示传值运算,在 f(a)函数中修改了 a 的值,但并不影响全局变量 a,其输出的值是 1。在右边表示传地址,形式参数是一个列表。在函数 f(a)内部修改其第一个单位的值,在函数外面 a 列表的值也随之修改。实际上,在 Python 中,不仅是参数传递存在这个问题。不同变量之间的复制,也存在类似的问题。在单元测试过程中,重点检查变量的传递是否符合规范。

程序 1-4　Python 传值和传地址比较

```
a=1
def f(a):
    a+=1
f(a)
print a
```

```
a=[1]
def f(a):
    a[0]+=1
f(a)
print a
```

2. 集成测试

集成测试是单元测试的下一个阶段，是指将通过测试的单元模块组装成系统或子系统，再进行测试，确保各模块集成在一起后能够符合设计要求，并确保增量的行为正确。集成测试的目标如下。

(1) 验证接口是否与设计相符。

(2) 发现设计和需求中存在的错误。

每一个单元模块并不是一个独立的程序，在测试一个模块时，可能和被测模块相联系的其他模块还没有完成开发。在这种情况下，需要用辅助模块去模拟与被测模块相联系的其他模块。这些辅助模块包括驱动模块和桩模块。驱动模块：是调用被测模块的辅助模块，它接收测试数据，把这些数据传送给被测模块，最后输出实测结果。桩模块：也称存根模块，用以代替被测程序调用的子模块。被测模块、驱动模块和桩模块共同构成了测试环境。

常见的集成测试策略包括：

(1) 自顶向下集成；

(2) 自底向上集成；

(3) 三明治集成。

自顶向下的单元测试策略：先对最顶层的单元进行测试，把顶层所调用的单元做成桩模块。其次对第二层进行测试，使用上面已测试的单元做驱动模块。以此类推直到测试完所有模块。优点是节省驱动函数的开发工作量，测试效率较高。

自顶向下的集成策略又包括深度优先和广度优先两种。

假设一个系统的结构如图 1-23(a)所示，包含 A、B、C、D、E 和 F 等 6 个模块，其中 B 调用 E，C 调用 F，A 调用 B、C、D 三个模块。图 1-23(b)演示了深度优先集成的过程，第 1 步为了测试模块 A，需要同时提供 B、C、D 三个桩模块 S_B，S_C，S_D，在测试模块 A 以后测试模块 B，需要提供模块 E 的桩模块 S_E，接下来测试模块 E，然后依次测试模块 C、F 和 D。若按照广度优先测试，那么其模块测试的次序如图 1-23(c)所示，分别为 A、B、C、D、E、F。

自底向上的单元测试策略：对模块调用层次图上最低层的模块进行单元测试，模拟调用该模块的模块做驱动模块。然后再对上面一层做单元测试，用下面已被测试过的模块做桩模块。以此类推，直到测试完所有模块。优点是节省桩函数的开发工作量。图 1-24 给出了一个自底向上集成示意图，假设系统共有 A、B、C、D、E 和 F 共 6 个模块，其中 B 调

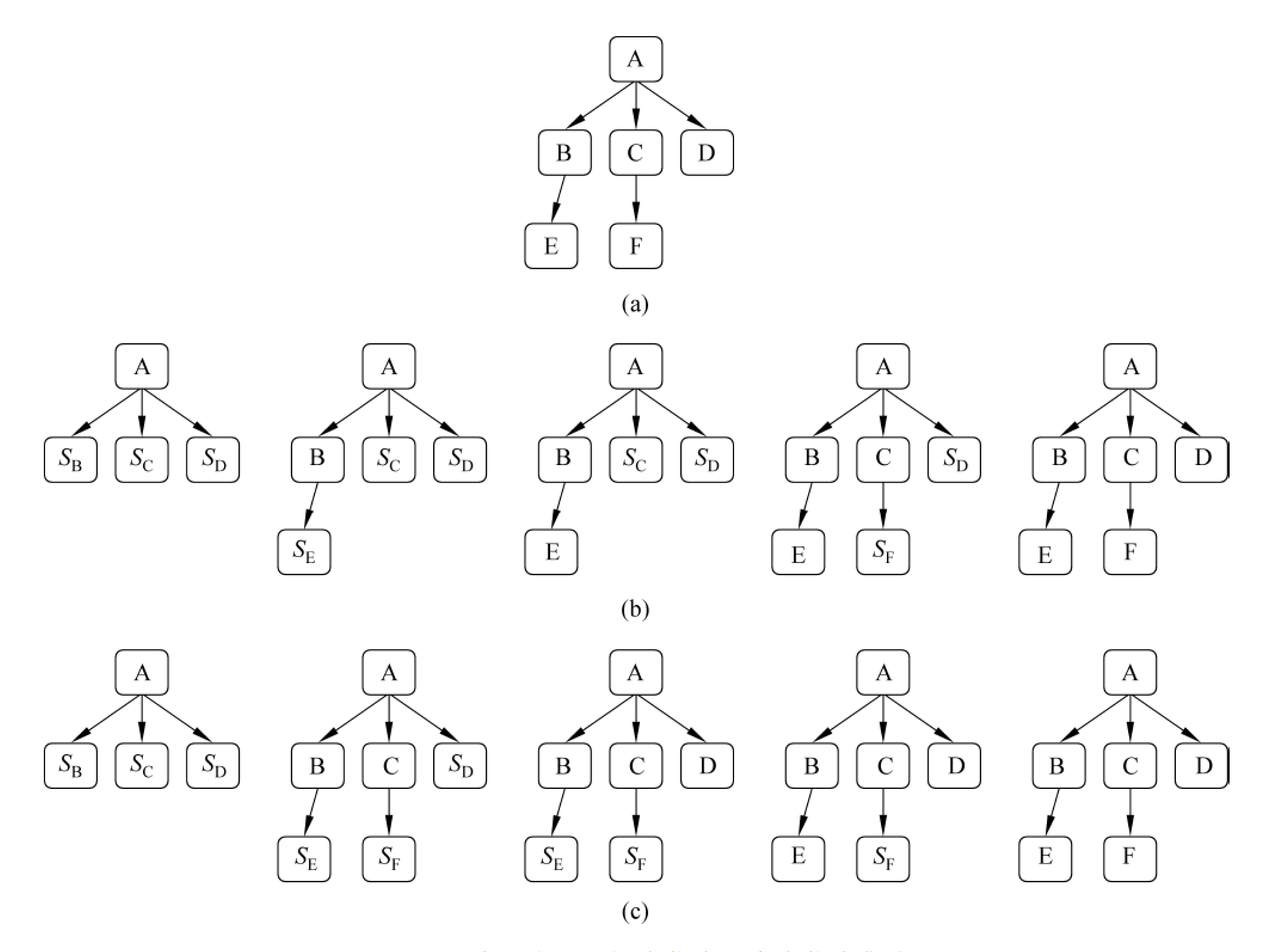


图 1-23 自顶向下的深度优先和广度优先集成

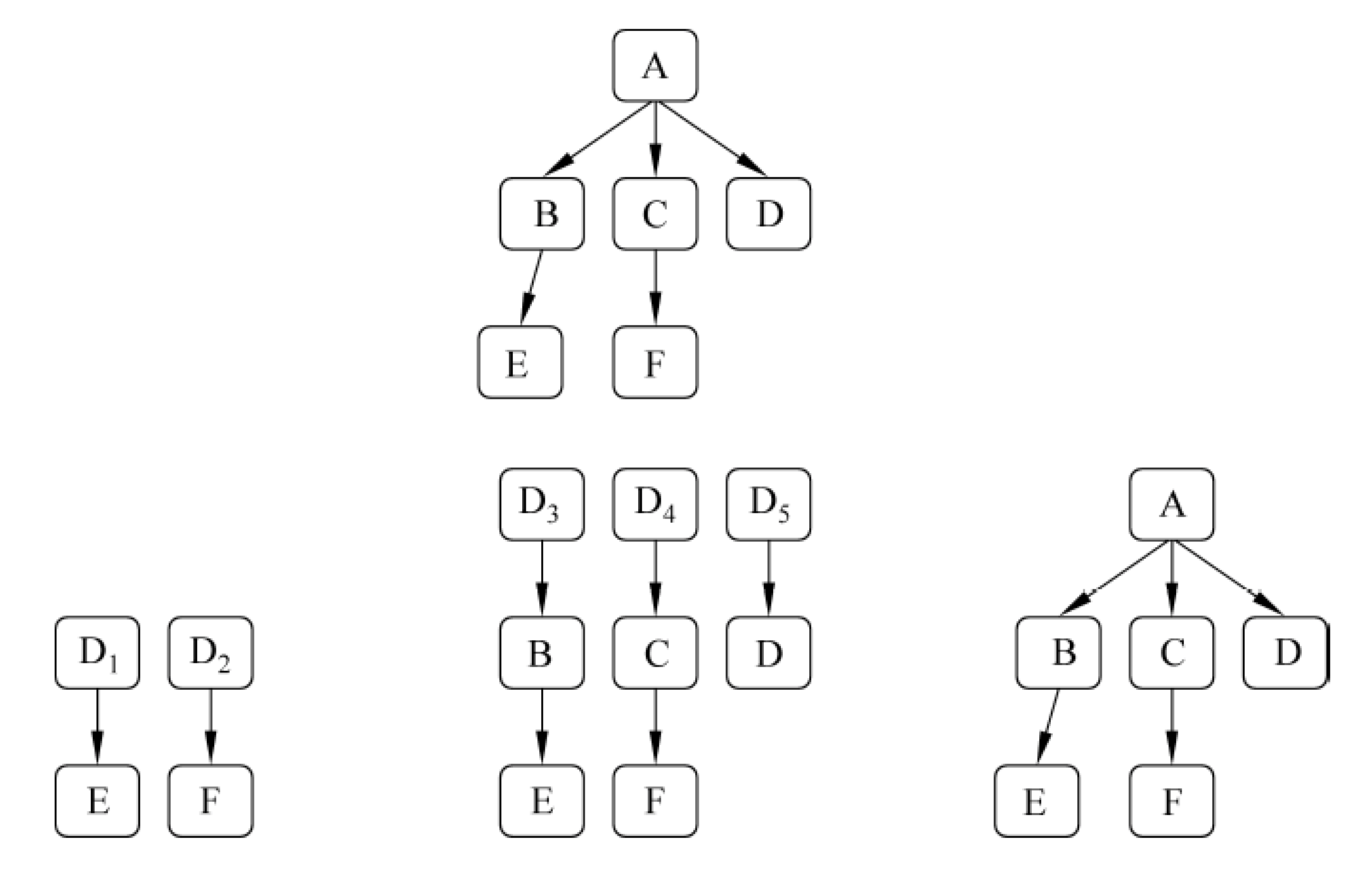


图 1-24 自底向上集成示意图

用 E，C 调用 F、A 调用 B、C、D 三个模块。先编写 E 和 F 的驱动模块，然后编写 B、C、D 的驱动，然后集成模块 A。

三明治集成：同时从顶层和底层向中间进行集成。

3. 系统测试

系统测试指将整个软件系统整体进行测试，在软件所运行的软硬件环境中进行测试，测试内容包括整个软件质量特性。由测试工程师在整个系统集成完毕后进行测试，前期主要测试系统的功能是否满足需求，后期主要测试系统的性能是否满足需求，以及系统在不同的软硬件环境中的兼容性。系统测试需要花大量的时间和精力去完成，是软件交付给用户进行验收测试的最后程序，其最终目的是为了确保软件产品能够被用户接受。

4. 验收测试

验收测试是指按照项目任务书或合同、供需双方约定的验收依据文档进行的对整个系统的测试与评审，用户决定是接收或拒收系统。验收测试完全采用黑盒测试技术，主要是用户代表通过执行其在平常使用系统时的典型任务来测试软件系统，根据业务需求分析，检验软件是否满足功能、行为、性能和系统协调性等方面的要求。只要有可能，在验收测试中就应该使用真实数据。在不使用真实数据的情况下，应该考虑使用真实数据的一个备份。备份数据的质量、精度和数据量必须尽可能地代表真实的数据。当使用真实数据或使用真实数据的备份时，仍然有必要引入一些手工数据，例如，测试边界条件或错误条件时，可创建一些手工数据。在创建手工数据时，测试人员必须采用正规的设计技术，使得提供的数据真正有代表性，确保软件系统能充分地测试。

国家标准GB/T 15532—2008《计算机软件测试规范》对于单元测试、集成测试、系统测试、验收测试的对象和目的、要求、内容、方法和过程都有详细的规定。

5. α测试和β测试

在开发的过程中，开发者无法预料用户将如何实际使用系统。如果软件是为多个用户开发的产品，让每个用户逐个执行正式的验收测试是不切实际的。很多软件产品生产者采用α测试和β测试的方法，发现可能只有最终用户才能发现的错误。

α测试是由用户在开发环境下进行的测试，也可以是公司内部的用户在模拟实际操作环境下进行的测试。α测试是在开发者的指导下，模拟各类用户行为对即将面市的软件产品(称为α版本)进行测试，试图发现并修改错误。α测试一般是产品达到一定的稳定性和可靠程度之后再开始。

经过α测试调整的软件产品称为β版本，β测试是由软件的多个用户在实际使用环境下的测试。与β测试不同的是，开发者通常不在测试现场，测试是在开发者无法控制的环境下进行的。在β测试中，由用户发现缺陷并定期向开发者报告，开发者对β版本做出修改，最后将软件产品交付给全体用户使用。α测试是β测试的基础。

第 2 章　传统的黑盒测试

黑盒测试是将软件看成一个黑盒子，测试人员无须考虑其内部逻辑结构，根据规格说明书设计测试用例。黑盒测试重点关注软件实现和规格说明书的一致性。根据输入/输出确定的逻辑关系设计测试数据，以检查其是否正确。常见的黑盒测试包括边界值分析、等价类划分、决策表和因果图。大量实践表明，软件缺陷出现在边界的几率比内部要大些。在设计测试用例时重点分析其输入和输出的边界情况，便产生了边界值分析法。人们将输入数据划分为不同的集合，并且假设在同一个输入集合内的元素发现错误的能力是等价的，等价类划分基于这个假设而设计。决策表和因果图都是基于输入和输出的逻辑关系构造测试用例。本章重点介绍这些黑盒测试方法。

2.1　边界值分析

2.1.1　边界值分析概念

长期的软件测试工作经验证明，大部分的缺陷是发生在输入或者输出的边界条件上，而不是内部。边界值分析是一种最常用的黑盒测试方法之一。例如，在编写算法时，循环语句的循环次数很容易出错，容易多写一次循环或者少写一次循环。例如，要求循环次数是 10 次，但是如果将循环错误地写成 for i in range(1,10)，会使循环少一次，导致软件出错。实践表明，在测试中考虑输入输出的边界是有必要的，它具有更高的测试效率和测试回报率。

常见的边界包括两种：语言相关的边界和业务相关的边界。

1. 语言相关的边界

当定义不同类型的数据时，其取值范围是不一样的。如果在定义变量时，没有充分考虑所使用数据的范围，那么就可能出现错误。

表 2-1 给出以 Java 语言中的整型数据类型取值范围的例子，显然在进行代码编写时，必须考虑这个变量将来可能出现的边界是否和变量的类型相一致。由于在 Java 语言中，必须显式定义变量类型，然后执行赋值动作，在定义变量时必须考虑变量的取值范围。

表 2-1　Java 语言整数的取值范围

类型	长度/b	范　　围
byte	8	$-2^7 \sim 2^7-1$
short	16	$-2^{15} \sim 2^{15}-1$
int	32	$-2^{31} \sim 2^{31}-1$
long	64	$-2^{63} \sim 2^{63}-1$

Python 是在首次赋值时决定该变量的类型。如果第一次赋值的变量恰好达到该变量所要求的长度，那么其能够获得正确的变量类型，否则也和 Java 语言一样可能会存在错误。Python 的标准整数类型是最通用的数字类型。在大多数 32 位机器上，标准整数类型的取值范围是 $-2^{31} \sim 2^{31}-1$，也就是 $-2\,147\,483\,648 \sim 2\,147\,483\,647$。如果在 64 位机器上使用 64 位编译器编译 Python，那么整数的长度将是 64 位。

2. 业务相关的边界

业务相关的边界是由业务特性所决定的边界。在这些边界上的数值，往往具有特定的要求，如果没有合适处理，就会导致错误。

边界的条件是由业务自身所决定的，不同的业务会产生不同的业务边界。例如，学生考试成绩的范围一般在 0～100 分之间，那么其下边界为 0，上边界为 100。如果规格说明中规定："重量在 10～50kg 范围内的邮件，其邮费计算公式为……"在这个需求规格说明书中，10kg 和 50kg 分别为其上边界和下边界。

边界值的取值会根据输入的范围区间不同而发生改变，分析边界值可以用图示分析法。图示分析法用图的形式表示出输入值的边界情况，包括三种类型的点：上点、离点和内点。上点是指边界上的点，离点是指距离上点最近的点。如果输入区间是开放的，则离点在区间内部，如果输入区间是封闭的，则离点在区间外部。内点是在区间内部的任意点。用不同的图标标识上点、离点和内点，可以更加清晰地看到边界的周边取值，测试用例取值时可以参考这些点。

用黑色实点表示有效的上点，白色实点表示无效的上点，内部有斜线的点表示离点，内部有点的点表示内点，如图 2-1 所示。图 2-1 表示输入值的取值范围是半开半闭区间 $(a,b]$，由上点、离点和内点的定义可知：a 是无效的上点，b 是有效的上点，c 和 e 是离点，d 是内点。

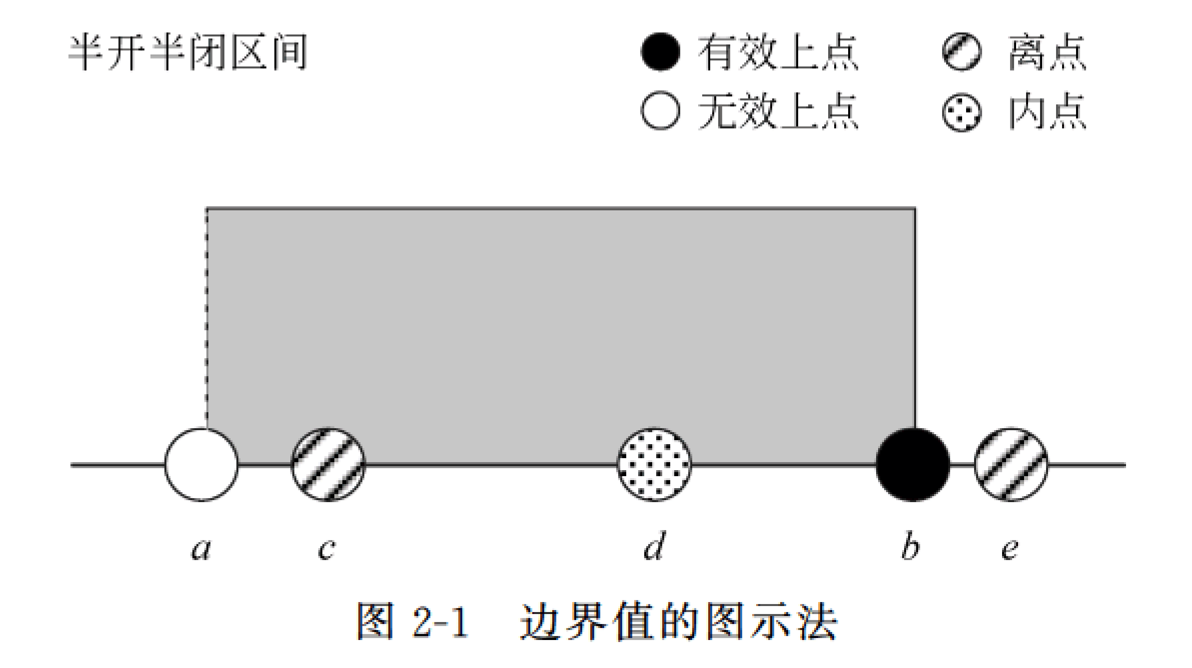


图 2-1 边界值的图示法

2.1.2 边界值分析原则

边界值的确定不仅要考虑输入数据，还要考虑输出结果。边界值分析设计测试用例时，进行边界值划分通常有以下几条规则。

(1) 如果输入值有确定的范围并且是连续的，则测试数据可以取最小值、略大于最小

值、正常值、略小于最大值以及最大值进行测试。例如，函数 $y=5x+2$ 的定义域是 $[0,3]$，则测试数据 x 可以取 $-1,0,0.1,2,2.9,3$ 以及 3.1。

(2) 如果输入值有确定的范围并且是离散的，则测试数据可以取该离散范围内存在最小值、略大于最小值、正常值、略小于最大值以及最大值进行测试。

(3) 如果输入数据有特殊的结构，则可以根据该结构的特性进行测试用例的设计，比较灵活。例如，输入的数据是一份文件，则可以取文件开头和结尾的数据来进行测试。

(4) 如果程序的规格说明给出的输入/输出域是有序集合(例如有序表、顺序文件等)，则应选取集合中的第一个元素以及最后一个元素作为测试用例。

(5) 分析规格说明，找出其他可能的边界条件。

2.1.3 边界值确定和分析法

边界值分析法中最重要的部分是如何确定边界值。对软件的输入进行边界值分析，其原理和分析函数定义域的边界类似，从不同的输入参数个数(函数参数个数)、输入与输出的逻辑关系(函数映射关系)以及输出条件(函数值域)几个方面进行分类讨论。下面采用图示分析方法，确定测试用例的输入边界。

1. 一个输入参数

若程序只包含一个输入参数，记为 x。参数 x 的取值区域是一个一维的空间，x 在这个空间占有一定的区间。在寻找输入参数 x 的边界值时主要考虑三个因素：①区间是开区间还是闭区间；②区间是否存在界；③在内部是否都存在间断点。

情况一：输入参数 x 的取值区域为闭区间 $[a,b]$，如图 2-2 所示。边界值是 a 和 b，并且都是有效值。

闭区间的取值如下。

(1) 上点：a，b。

(2) 离点：d，e(分别无限接近点 a 和点 b)。

(3) 内点：c。

测试用例的 x 取值应该为 d、a、c、b 和 e 这 5 个点的值，即包括上点、离点和内点。

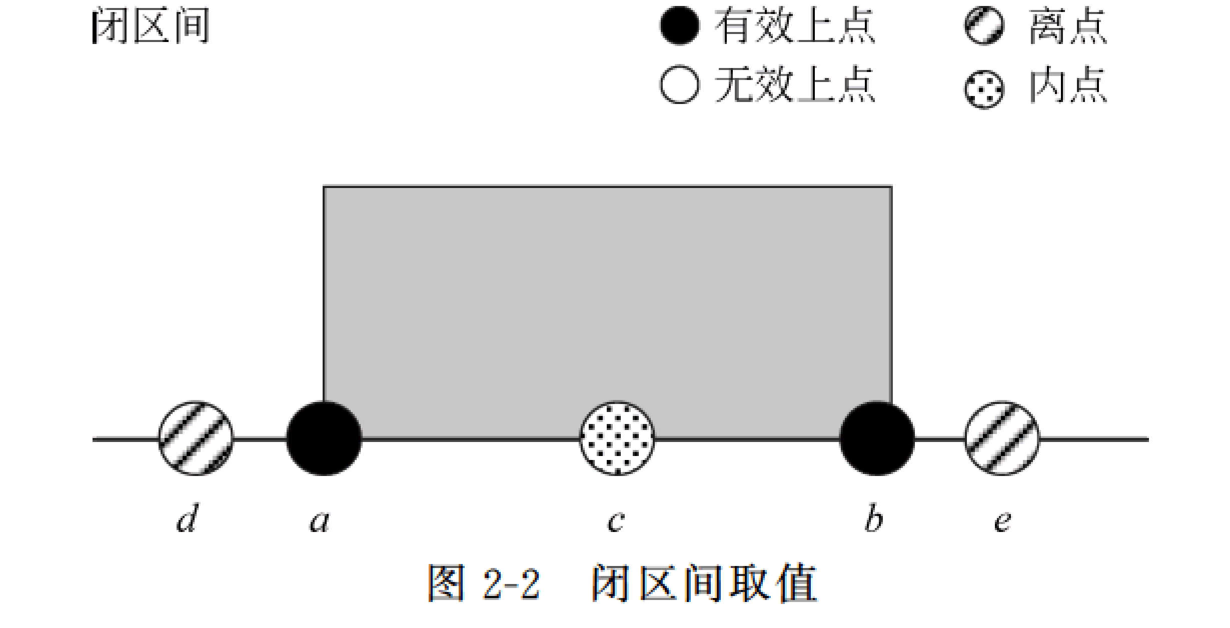


图 2-2 闭区间取值

情况二：输入参数 x 的取值区域为开区间 (a,b)，如图 2-3 所示。边界值是 a 和 b，并且都是无效值。

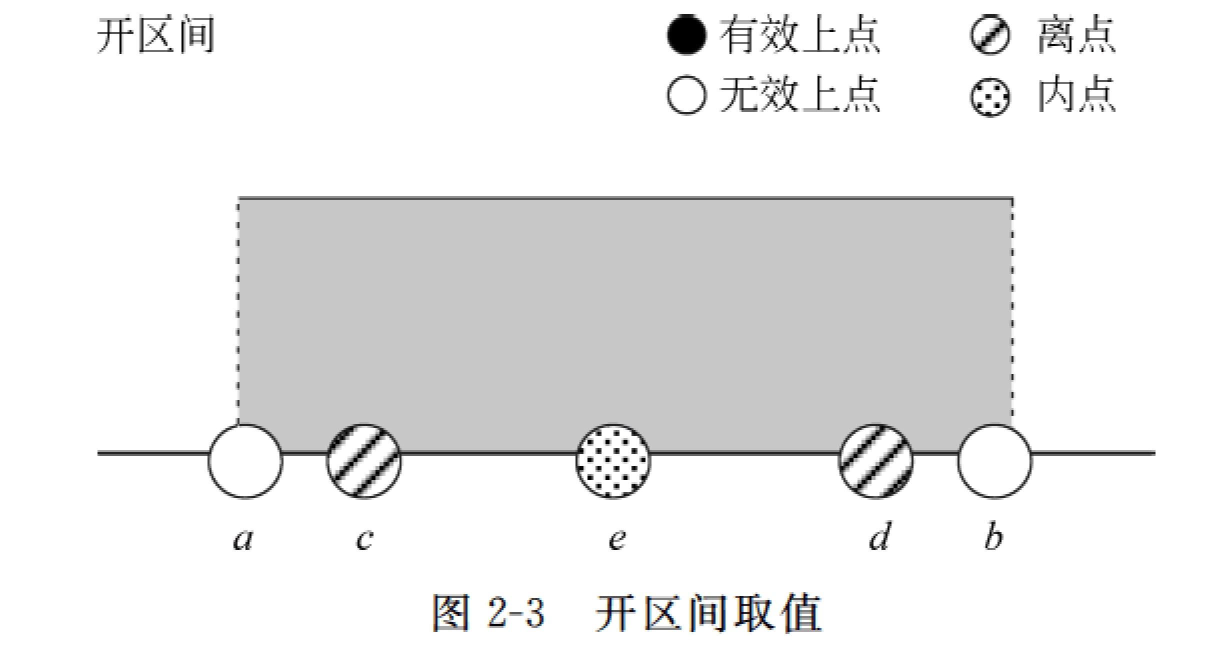


图 2-3 开区间取值

开区间的取值如下。

(1) 上点：a，b。

(2) 离点：c，d（分别无限接近点 a 和点 b）。

(3) 内点：e。

测试用例的 x 取值应该为 a、c、e、d 和 b 这 5 个点的值，即包括上点、离点和内点。

情况三：输入参数 x 的取值区域为半开半闭区间 $(a,b]$，如图 2-4 所示。边界值是 a 和 b，其中 a 是无效上点，b 是有效上点。

半开半闭区间的取值如下。

(1) 上点：a，b。

(2) 离点：c，e（分别无限接近点 a 和点 b）。

(3) 内点：d。

测试用例的 x 取值应该为 a、c、d、b 和 e 这 5 个点的值，即包括上点、离点和内点。

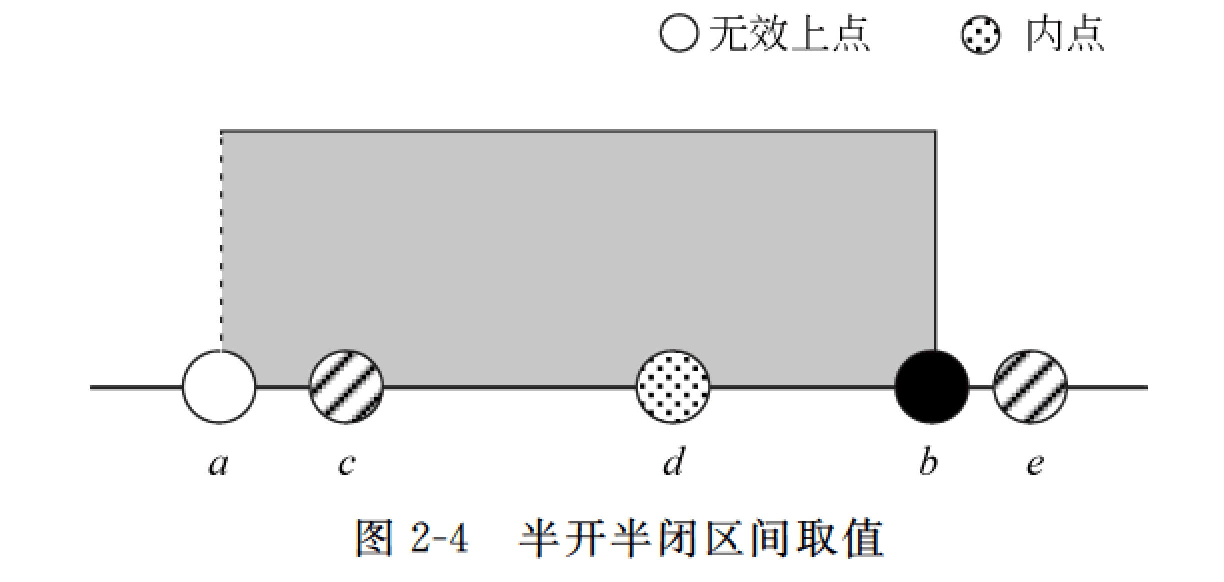


图 2-4 半开半闭区间取值

情况四：输入参数 x 的取值区域为无界区间 $[a,+\infty]$，如图 2-5 所示。边界值是 a，并且是有效的，另一个边界是正无穷大。实际测试的时候可以选择一个足够大的数来代替无穷大，图中用 d 表示。

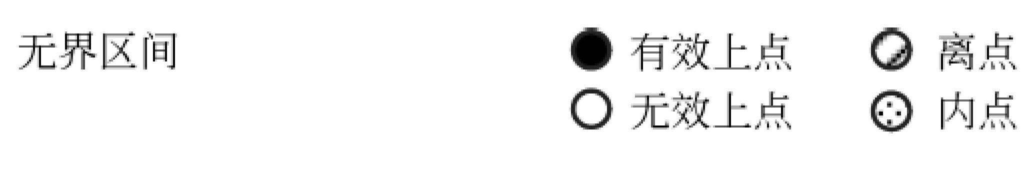
无界区间的取值如下。

(1) 上点：a，d。

(2) 离点：b（无限接近点 a）。

(3) 内点：c。

测试用例的 x 取值应该为 b、a、c 和 d 这 4 个点的值，即包括上点、离点和内点。



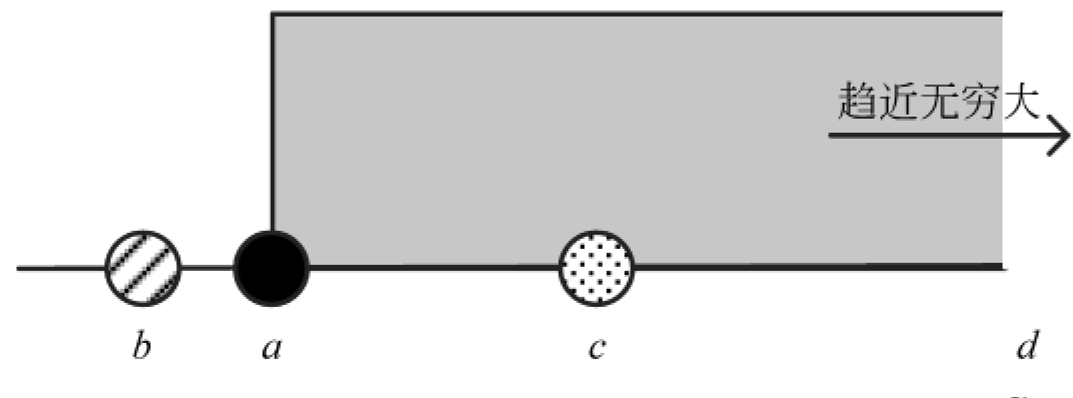


图 2-5　无界区间是$[a,+\infty]$的情况

情况五：输入参数 x 的取值区域不是一个连续的区域，而是多个连续的区间。假设输入域是两个区间的并集$[a,b]\cup[c,d]$，如图 2-6 所示。则边界值是 a、b、c 和 d，并且都是有效的。

多个连续区间取值如下。

(1) 上点：a,b,c,d。

(2) 离点：e,g,h,j。

(3) 内点：f,i。

测试用例的 x 取值应该为 e、a、f、b、g、h、c、i、d 和 j 这 10 个点，即包括上点、离点和内点。不能只考虑 a 和 d 这两个最小值和最大值边界，而应该考虑所有连续区间的边界。

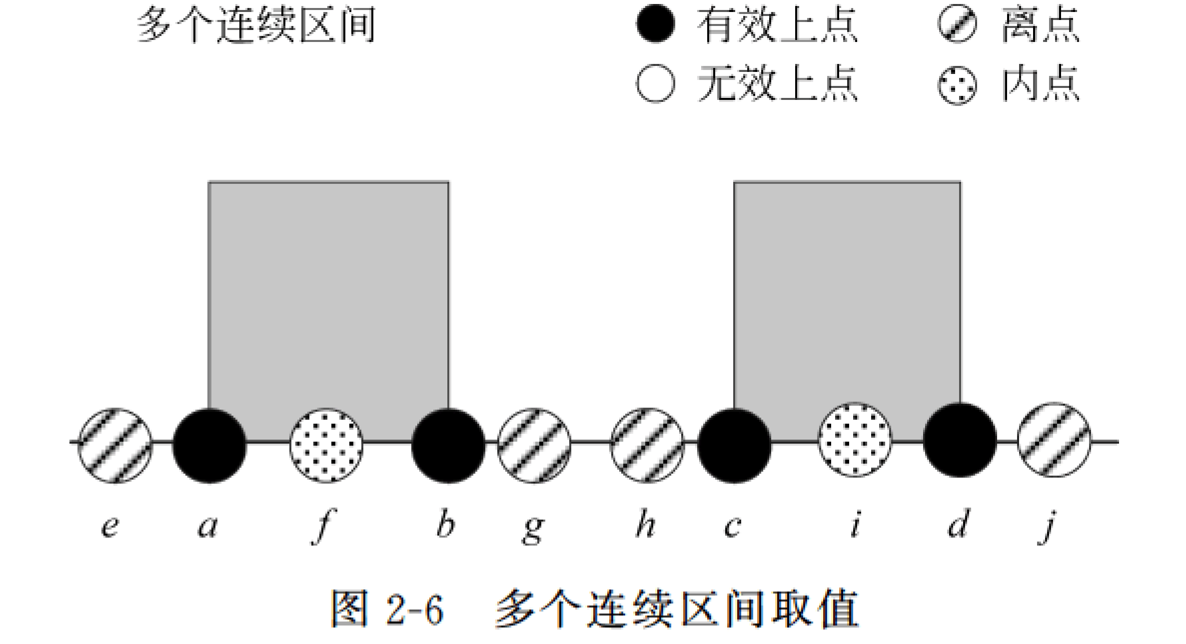


图 2-6　多个连续区间取值

情况六：输入参数 x 的取值区域为区间$[a,b]$，其中有一个点 c 是间断点，那么实际区间可表示为$[a,c)\cup(c,d]$，如图 2-7 所示。这里的 c 是一个间断点，可以认为 c 把一个连续的区域划分成了两个独立的区域，并且 c 是这两个区域的边界。这种情况下的边界值是 a、b 和 c，其中 a 和 b 是有效的，c 是无效的。

有间断点区间取值如下。

(1) 上点：a,b,c。

(2) 离点：d,e,f,g。

(3) 内点：h,i。

测试用例的 x 取值应该为 d、a、h、f、c、g、i、b 和 e 这 9 个点，即包括上点、离点和内点。不能只考虑 a 和 b 这两个边界。

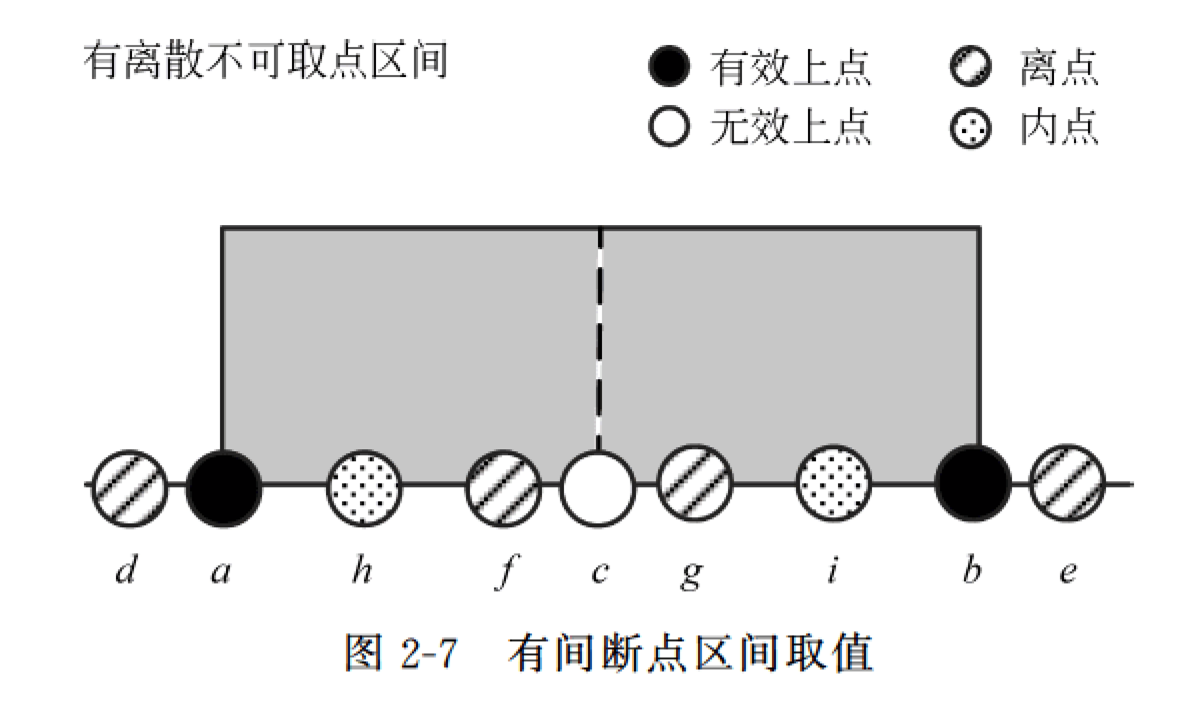


图 2-7　有间断点区间取值

分析一个输入参数的边界时，首先根据产品规格说明书找到输入参数的边界，然后看这个边界点是否可取。闭区间的边界都是可取的，开区间的边界是不可取的，半开半闭区间的一个边界可取，另外一个边界不可取。如果边界是无穷大，则可以取一个足够大的值作为边界。如果有间断点，可以认为间断点把一个完整的区间分成两个子区间，并且这个间断点是不可取的。一个参数若有多个边界，在设计测试用例时需要考虑所有的边界情况。

2．两个输入参数

两个输入参数，分别记为 x 和 y，则输入域是由 x 和 y 共同确定的一个平面区域，围成该区域的直线或者曲线则是区域边界。在各个边界上的值可能是有效值，也可能是无效值。值得注意的是，只有一个输入参数时，只需要考虑一个参数的取值范围。当有两个输入参数时，不仅要考虑单个输入参数的取值范围，还要考虑两个参数之间的共同约束。

例如，输入参数 x 的取值区间是[1,3]，y 的取值区间也是[1,3]，同时 x 和 y 需要满足方程 $x+y<1$，那么在考虑边界时，首先要考虑 x 和 y 各自的边界，都是 1 和 3。同时要考虑两者之间共同约束的边界 $x+y<1$。由于在分析一个输入参数时已经讨论过开闭区间、间断点和边界是否无界，两个输入参数就不再重点讨论这三个因素，主要针对共同约束展开讨论。

情况一：输入域是一些离散点，随机分布在二维空间中，如图 2-8 所示。这种情况下可以分别单独确认输入参数 x 和 y 的边界，需要找到所有离散点中 x 的最小值和最大值以及 y 的最小值和最大值。但是，这样会忽略 x 和 y 的一些潜在联系。如果要将所有情况都考虑进去，可以单独分析每个离散点。当离散点较少的时候测试比较方便，但当有许多离散点时会很费时间，而且效率很低。所以需要一种折中的方法，具体如下：将离散点最外面的点用直线相连，如图 2-9 所示，连起来的线段表示的就是这个输入域的边界。这可以很好地提高测试的效率，同时可以较好地表示离散点的取值范围。

离散点区域的取值如下。

(1) 上点：线段 ab、bd、cd 和 ac 上的点(只有点 a、b、c 和 d 是有效值，其他都是无效值)。

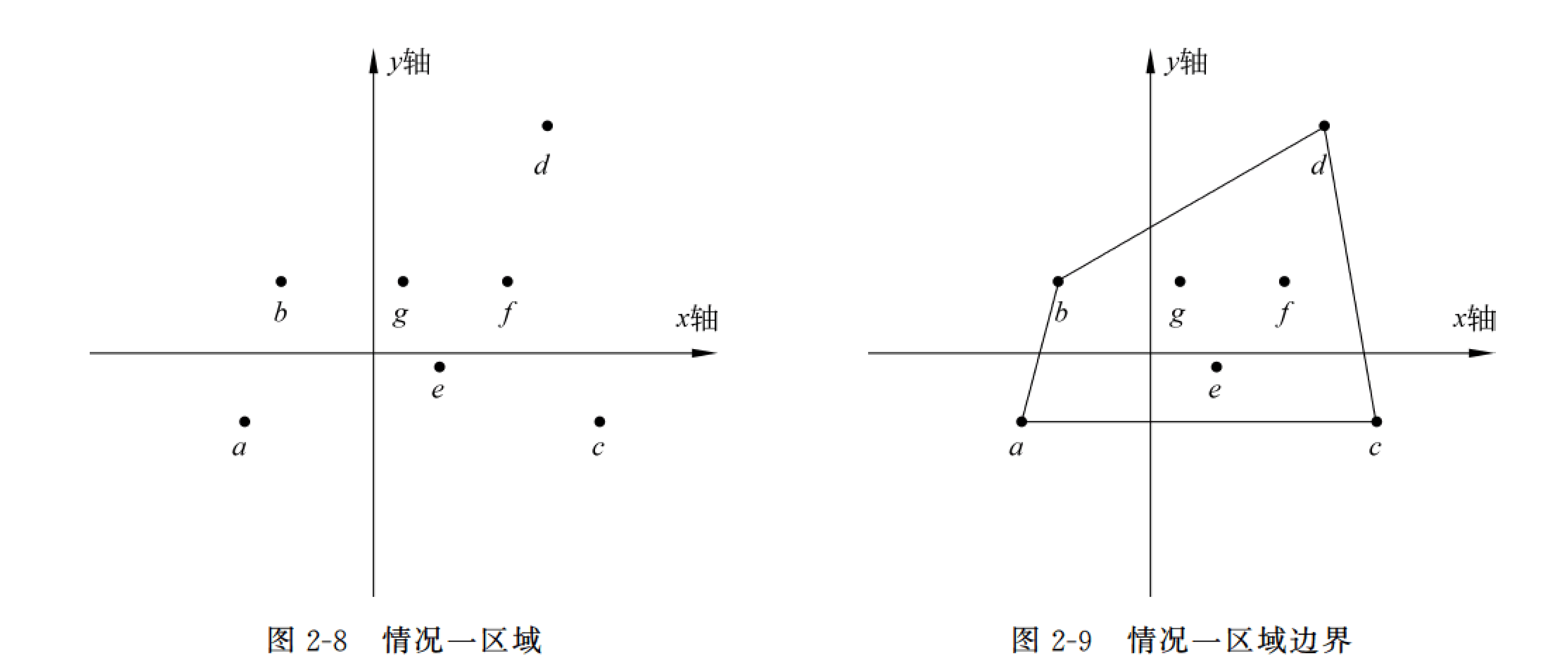


图 2-8 情况一区域

图 2-9 情况一区域边界

(2) 离点：线段 ab、bd、cd 和 ac 围成区域内的点(原本就存在的离散点是有效值，其他都是无效值)。

(3) 内点：线段 ab、bd、cd 和 ac 围成区域外的点。

这种方法也存在一定的缺陷，例如有一个点离所有的点很远，在连接最外面的点时会有很大的一部分内部区域是无效的，边界就变得没有意义，这时需要将这个点单独取出。所以，遇到具体问题还要再分析，这里只是提供一种基本的方法。

确定边界后，测试用例设计方法和两个输入参数的情况三一致，这里不再具体说明怎样设计测试用例，在情况三中详细说明。

情况二：输入域是二维空间中的曲线或者直线，只有在这些线上的取值是有效的。图 2-10 中只有在圆周上的取值是有效值。首先单独分析输入参数 x 和 y 的边界，x 的取值区间是 $[a,b]$，y 的取值区间是 $[c,d]$。如果不深入考虑 x 和 y 之间的约束，就会得到 4 个边界 $x=a$，$x=b$，$y=c$ 和 $y=d$，这是一个矩形，如图 2-11 所示。但是实际的边界是圆周，比上面 4 个边界确定的矩形边界要小很多。

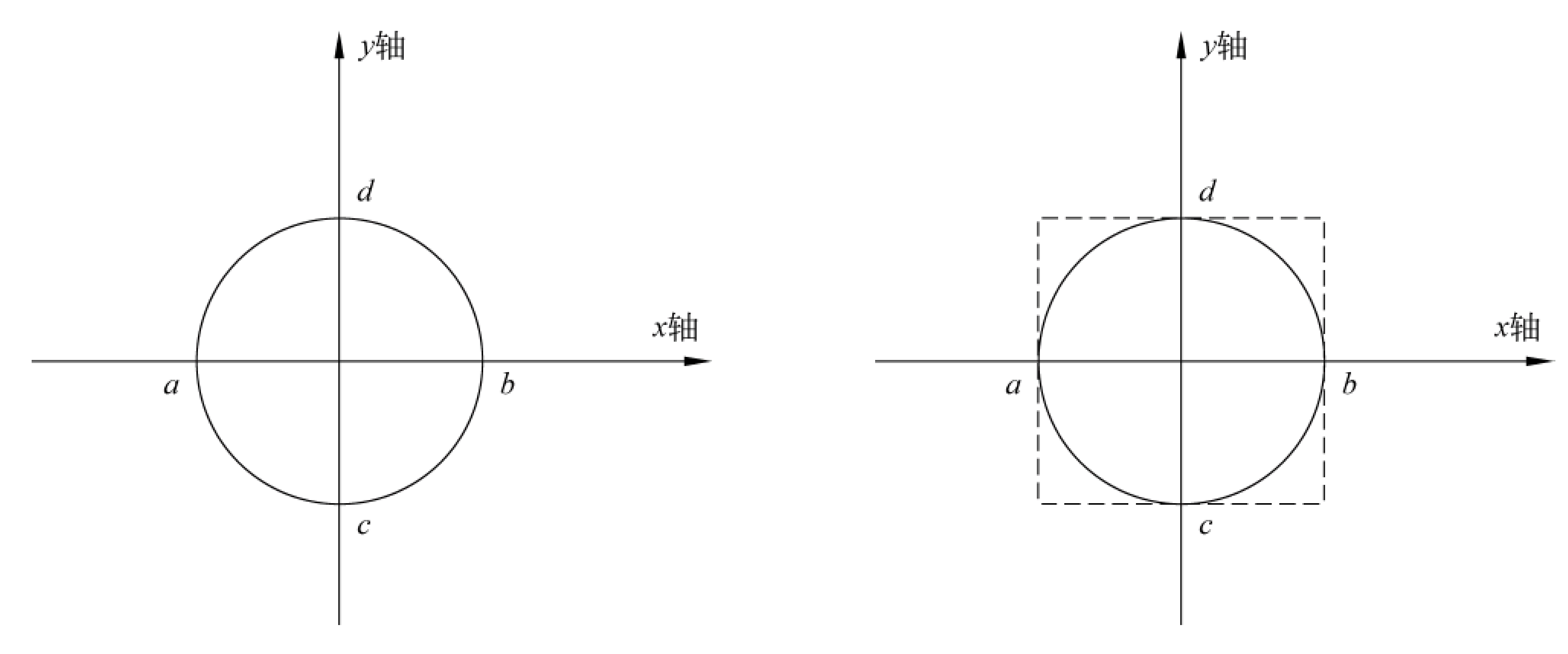


图 2-10 曲线或者直线输入域

图 2-11 不考虑约束的边界

曲线或者直线区域的取值如下。

(1) 上点：圆周上的点。

(2) 离点：无限接近圆周的点(包括圆内和圆外的点)。

(3) 内点：无(没有边界内的点)。

输入域是一条曲线或者直线，边界也就是这条线。在设计测试用例时，要考虑边界上的边界点，而不是仅考虑 x 和 y 的各自边界的组合值。图 2-10 中的边界点是 $(a,0)$、$(b,0)$、$(0,c)$ 和 $(0,d)$，所以可以取这些点以及附近的点，具体的测试用例设计如表 2-2 所示。

表 2-2　情况二对应测试用例

用例编号	x	y	预期输出	用例编号	x	y	预期输出
1	$a-1$	0	边界外	7	0	$d-1$	边界外
2	a	0	边界上	8	0	d	边界上
3	$a+1$	0	边界外	9	0	$d+1$	边界外
4	$b-1$	0	边界外	10	0	$c-1$	边界外
5	b	0	边界上	11	0	c	边界上
6	$b+1$	0	边界外	12	0	$c+1$	边界外

情况三：输入域是二维空间中的面，并且该面上的取值都是有效的，没有不可取的值，如图 2-12 所示。图中输入参数的取值区间是一个圆，而且圆内部的值都是有效的。这和情况二不同，情况二只有曲线或者直线上的值是有效的，情况三边界内的也是有效的。和情况二的分析过程类似，首先可以分别找出 x 和 y 的取值区间。x 的取值区间为 $[a,b]$，y 的取值区间为 $[c,d]$，同时区域边界是 x 和 y 的共同边界，需要满足一定的约束条件，图中显示边界是圆的圆周。

取值如下。

(1) 上点：圆周上的点。

(2) 离点：无限接近圆周且在圆外的点。

(3) 内点：圆内的点。

图 2-12　情况三区域

在设计测试用例时，输入域是一个面和输入域是一条线既有相同的地方又有不同的地方。相同的是输入域是一个面的边界还是一个曲线或者直线；不同的是输入域是一个面，需要考虑边界内的正常值。针对边界，测试用例的设计和情况二相同。在这个基础上还要增加边界内的正常值，例如 $(0,0)$。这和边界没有关系，但是也要进行测试。具体的测试用例设计如表 2-3 所示。

表 2-3 情况三对应的测试用例

用例编号	x	y	预期输出	用例编号	x	y	预期输出
1	$a-1$	0	边界外	8	0	d	边界上
2	a	0	边界上	9	0	$d+1$	边界外
3	$a+1$	0	边界内	10	0	$c-1$	边界外
4	$b-1$	0	边界内	11	0	c	边界上
5	b	0	边界上	12	0	$c+1$	边界内
6	$b+1$	0	边界外	13	0	0	边界内
7	0	$d-1$	边界内				

情况四：输入域是二维空间中的面，这个面的内部有无效值，如图 2-13 所示。图中显示的是一个环，这时候就不仅存在外边界，还存在内边界。外边界 x 的取值范围为 $[a,b]$，y 的取值范围为 $[e,f]$；内边界 x 的取值范围为 $[e,f]$，y 的取值范围为 $[h,g]$。可以发现，在边界上，一个 x 值可能对应一个 y 值、两个 y 值和 4 个 y 值，例如 x 取 0 时，边界上 y 的取值有 4 个，分别是 d、g、h 和 c。

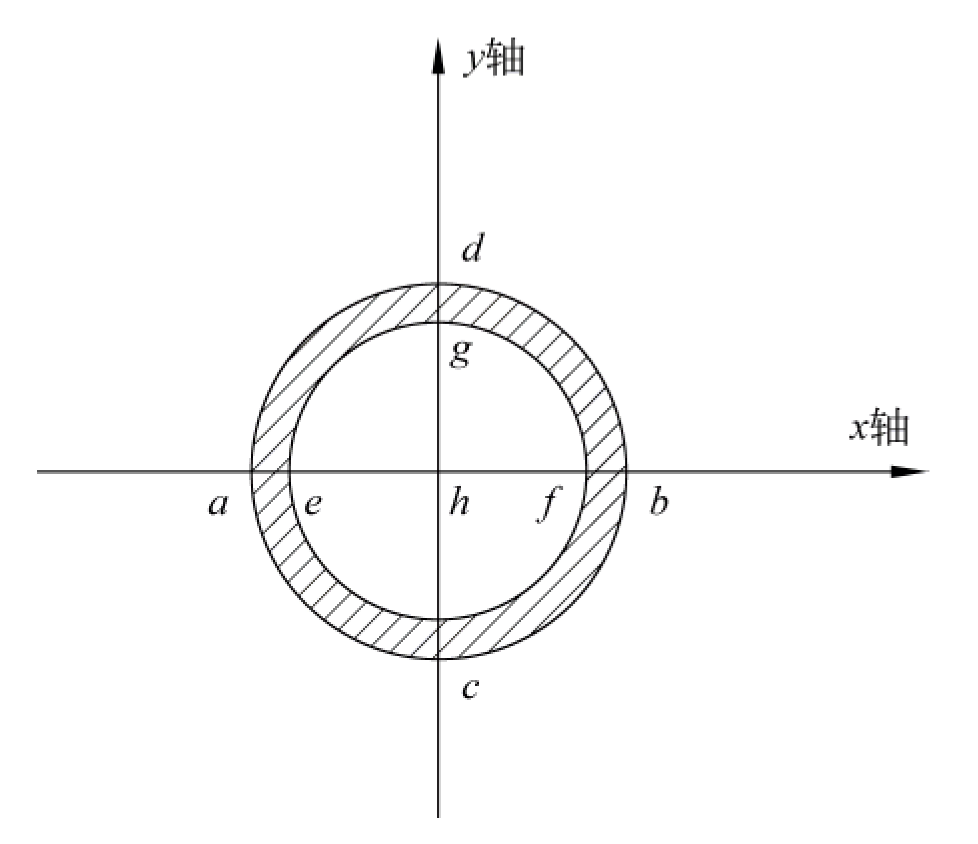
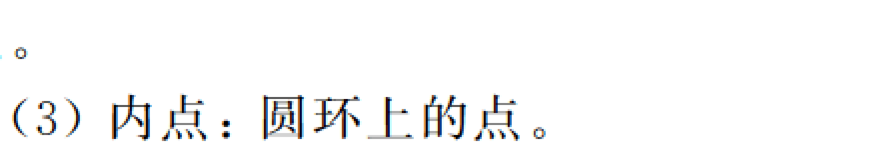


图 2-13 情况四区域

取值如下。

(1) 上点：内圆周和外圆周上的点。



(2) 离点：内圆周以内和外圆周以外的点。

(3) 内点：圆环上的点。

这种情况和情况三的测试用例设计方法基本一样，在考虑的时候要把外边界和内边界都考虑进去，同时还需考虑两个边界之间的点。具体的测试用例设计如表 2-4 所示。

表 2-4 情况四对应的测试用例

用例编号	x	y	预期输出	用例编号	x	y	预期输出
1	$a-1$	0	边界外	11	0	$g-1$	边界外
2	a	0	边界上	12	0	g	边界上
3	$(a+e)/2$	0	边界内	13	0	$(g+d)/2$	边界内
4	e	0	边界上	14	0	d	边界上
5	$e+1$	0	边界外	15	0	$d+1$	边界外
6	$f-1$	0	边界外	16	0	$c-1$	边界外
7	f	0	边界上	17	0	c	边界上
8	$(f+b)/2$	0	边界内	18	0	$(c+h)/2$	边界内
9	b	0	边界上	19	0	h	边界上
10	$b+1$	0	边界外	20	0	$h+1$	边界外

总结两个输入参数的情况可以得出,输入域的形态判别很重要。在实际测试中,可以利用各种工具(Matlab、MathStudio 等)分析输入参数的形态,然后根据具体的形态来分析其边界值。

3. 三个输入参数

三个输入参数的情况相对于前面分析的两类情况更加复杂,它的输入域是在一个三维的空间中,可能是点、线、平面,也可能是一个三维的立体空间。如果用 x_1,x_2,x_3 分别表示三个输入参数,则输入域可表示为 $D=\{(x_1,x_2,x_3)|P(x_1,x_2,x_3)\}$。当输入域是一个三维的立体空间时该如何确定边界? 首先,根据输入域的定义分析得到三个输入参数的边界。其次,找到各个参数之间的联系或者约束。三维的立体空间通常是由面围成的,这些面就是输入参数的边界。例如,输入域是一个立体的三角形,则边界就是围绕这个三角形的三个三角平面。在测试时,每个边界(面)都要考虑。

4. 间接推断隐含边界

前面讨论的情况都是明确知道了输入域的直接约束条件(或者已经推断出来),并在此基础上去分析输入域的边界情况。实际测试中,测试人员可能不能直接从软件规格说明书中得出直接约束条件,这就需要找到隐含的关系才能推断分析得到输入域的边界。在分析边界时,需要一层层去分析输入与输出的逻辑关系,找到真正输入的边界,过程中可能会涉及许多中间参数。下面将从 4 种不同的情况分析如何间接推断边界。

第一种情况:输入参数和输出之间的逻辑关系未知,但有一个参数与输入输出都有逻辑关系,称其为中间参数。如果已知中间参数的约束条件,可根据中间参数和输入参数之间的逻辑关系求得输入参数的约束条件,再分析边界情况。

例如,有一个程序检查水桶质量,需计算圆柱形水桶的体积。已知水桶高度为 h,程序的输入参数是水桶底部半径 r,同时要求水桶底面积 S 不得大于 m 平方分米,则如何分析这个程序的输入边界?(公式用 $V=S\times h$。)

输入参数是半径 r,中间参数是底面积 S,输出是体积 V。水桶是圆柱体,则水桶底部是一个圆,底面积 $S=\pi r^2$。隐藏的约束条件是水桶底部半径 r 要大于 0,所以输入参数 r 的一个边界是 0。另外中间参数 $S\leqslant m$,则可推断得到输入参数半径 $r\leqslant\sqrt{(m/\pi)}$。可以看出 r 的另一个边界是 $\sqrt{(m/\pi)}$,并且是可取的,这是由中间参数 S 以及它们之间的逻辑推断得到的,如表 2-5 所示。

表 2-5　包含中间参数约束的示例

<table>
<tr><th>中间参数(底面积 S)</th><th colspan="2">输入参数(底面半径 r)</th></tr>
<tr><td>$S\leqslant m$</td><td>$r\leqslant\sqrt{(m/\pi)}$</td><td rowspan="2">$0<r\leqslant\sqrt{(m/\pi)}$</td></tr>
<tr><td>$S>0$(隐含条件)</td><td>$r>0$</td></tr>
</table>

第二种情况:已知输入参数和输出之间的逻辑关系,并且有另一个参数与输入参数有一定的逻辑关系,称其为输入参数的相关参数。根据相关参数和输入参数的逻辑关系,

可以推断出输入参数的约束条件，再进一步得到边界。

例如，同样以检查水桶质量程序为例，已知水桶高度 h，程序的输入参数改为底面积 S，并且要求半径 $r\leqslant m$，则如何分析这个程序的输入边界？（公式用 $V=S\times h$。）

输入参数是底面积 S，相关参数是半径 r，输出是体积 V。已知半径 $r\leqslant m$，计算 $S=\pi r^2\leqslant \pi m^2$，可得，输入参数 S 的约束条件是小于等于 $m^2\pi$ 平方分米。则输入参数 S 的一个边界是 $m^2\pi$，并且是可取的。这个推断过程用到了 S 的相关参数 r 的约束条件以及它们之间的逻辑关系，如表 2-6 所示。

表 2-6　包含相关参数约束的示例

相关参数(底面半径 r)	输入参数(底面积 S)	
$r\leqslant m$	$S\leqslant \pi m^2$	$0<S\leqslant \pi m^2$
$r>0$(隐含条件)	$S>0$	

第三种情况：输入参数和输出之间的逻辑关系未知，已知中间参数和输出之间的逻辑关系以及输入参数和中间参数的逻辑关系。不同于第一种情况，中间参数的约束条件未知，但是已知中间参数的相关参数约束条件。既然相关参数的约束条件已知，可以根据与中间参数的逻辑关系，得到中间参数的约束条件。然后再根据中间参数和输入参数的逻辑关系，得到输入参数的约束条件，最后得到输入边界。

同样以检查水桶质量程序为例，已知水桶高度 h，程序的输入参数为水桶底面积 S，并且要求水桶底部周长 $C\leqslant m$。则如何分析这个程序的输入边界？（公式用 $V=\pi r^2\times h$。）

输入参数是底面积 S，中间参数是半径 r，相关参数是周长 C，输出是体积 V。由约束条件周长 $C\leqslant m$ 分米，以及公式 $C=2\pi r$ 计算得到半径 $r\leqslant m/2\pi$。又根据公式 $S=\pi r^2$ 得到底面积 $S\leqslant m$^2/4π。则输入参数 S 的一个边界是 m^2/4π，且是可取的。C 的一个隐藏约束条件 $C>0$，可以推断出 $S>0$。这个推断过程用到周长的约束条件、周长与半径的逻辑关系和半径与面积的逻辑关系，如表 2-7 所示。

表 2-7　中间参数和输入输出均有关系

中间参数的相关参数(周长 C)	中间参数(底面半径 r)	输入参数(底面积 S)	
$C\leqslant m$	$r\leqslant \frac{m}{2\pi}$	$S\leqslant \frac{m^2}{4\pi}$	$0<S\leqslant \frac{m^2}{4\pi}$
$C>0$(隐含条件)	$r>0$	$S>0$	

第四种情况：输入参数和输出之间的逻辑关系已知，同时有两个输入参数的相关参数，记为相关参数 1 和相关参数 2。已知输入参数和两个相关参数之间的逻辑关系，两个相关参数的约束条件也都明确。既然相关参数的约束条件已知，可以根据与输入参数的逻辑关系，得到输入参数的约束条件。值得注意的是，输入参数受到两个相关参数约束。

同样以检查水桶质量程序为例，已知水桶高度 h，程序的输入参数为水桶底部半径 r，并且要求水桶底部周长 $C\leqslant m_1$，水桶底面积 $S\geqslant m_2$。则如何分析这个程序的输入边界？（公式用 $V=\pi r^2\times h$。）

输入参数是半径 r，相关参数1是底面积 S，相关参数2是周长 C，输出是体积 V。由约束条件周长 $C \leqslant m_1$ 分米、底部面积 $S \geqslant m_2$ 平方分米、公式 $C=2\pi r$ 和 $S=\pi r^2$ 计算得到半径 $\sqrt{\frac{m_2}{\pi}} \leqslant r \leqslant \frac{m_1}{2\pi}$。则输入参数 r 边界为 $\sqrt{\frac{m_2}{\pi}}$ 和 $\frac{m_1}{2\pi}$，且都是可取的。这个推断过程用到周长 C 和底部面积 S 的约束条件以及它们与半径 r 的逻辑关系，如表2-8所示。

表2-8 输入和输出关系推导约束关系

相关参数1(周长 C)	相关参数1(底面积 S)	输入参数(底面半径 r)	
$C \leqslant m_1$	—	$r \leqslant \frac{m_1}{2\pi}$	$\sqrt{\frac{m_2}{\pi}} \leqslant r \leqslant \frac{m_1}{2\pi}$
—	$S \geqslant m_2$	$r \geqslant \sqrt{\frac{m_2}{\pi}}$	
$C>0$(隐含条件)	—	$r>0$	
—	$S>0$(隐含条件)	$r>0$	

总结上面几种情况，可以发现输入参数总是与其他的参数相联系。对其他参数的约束本质上就是对输入参数的约束。可以通过输入参数与其他参数的逻辑关系推断出输入参数的边界。

5. 根据输出条件分析边界值

软件测试中经常会遇到这样的情况：规格说明书中对输入域没有进行过多的说明，但是对输出域有一定的说明，或者对输出结果有一定的预测。这种情况下可以根据输出条件来确定输入域。根据不同输出对应着不同的输入，如果已知输出的边界也就可以倒推出输入的边界值。例如，某银行的ATM自动取款机的余额为30 000人民币，即输出的上限是30 000，则用户取款的输入金额的上边界也是30 000。

2.1.4 边界值测试举例

以判断三角形形状的程序为例，根据上面的边界值确定和分析法来设计测试用例。

问题描述：

有一个三角形 ABC，根据角的大小来判断三角形的形状。在能组成三角形的前提下，三角形是锐角、直角还是钝角？输入参数是 $\angle A$、$\angle B$ 和 $\angle C$ 的角度。其中，$\angle A$、$\angle B$ 和 $\angle C$ 的取值范围都是 $(0^\circ, 180^\circ)$，0° 和 180° 都不可取。

由问题描述可知，$\angle A$、$\angle B$ 和 $\angle C$ 的和必须是 180° 才能构成三角形。现将输入参数简单记为变量 A、B 和 C，输入域是三维空间中的一个超平面，这个三角形面的方程表达式是：$A+B+C=180$，其中 $0<A<180$，$0<B<180$，$0<C<180$，如图2-14所示。超平面外面的边界不可取，是因为 A、B 和 C 不能取0。

根据输出(三角形的形状)，可以分为4个等价类，输出结果分别是钝角三角形、直角三角形、锐角三角形和非三角形。钝角三角形对应的输入条件是 $\angle A$、$\angle B$ 和 $\angle C$ 中间有

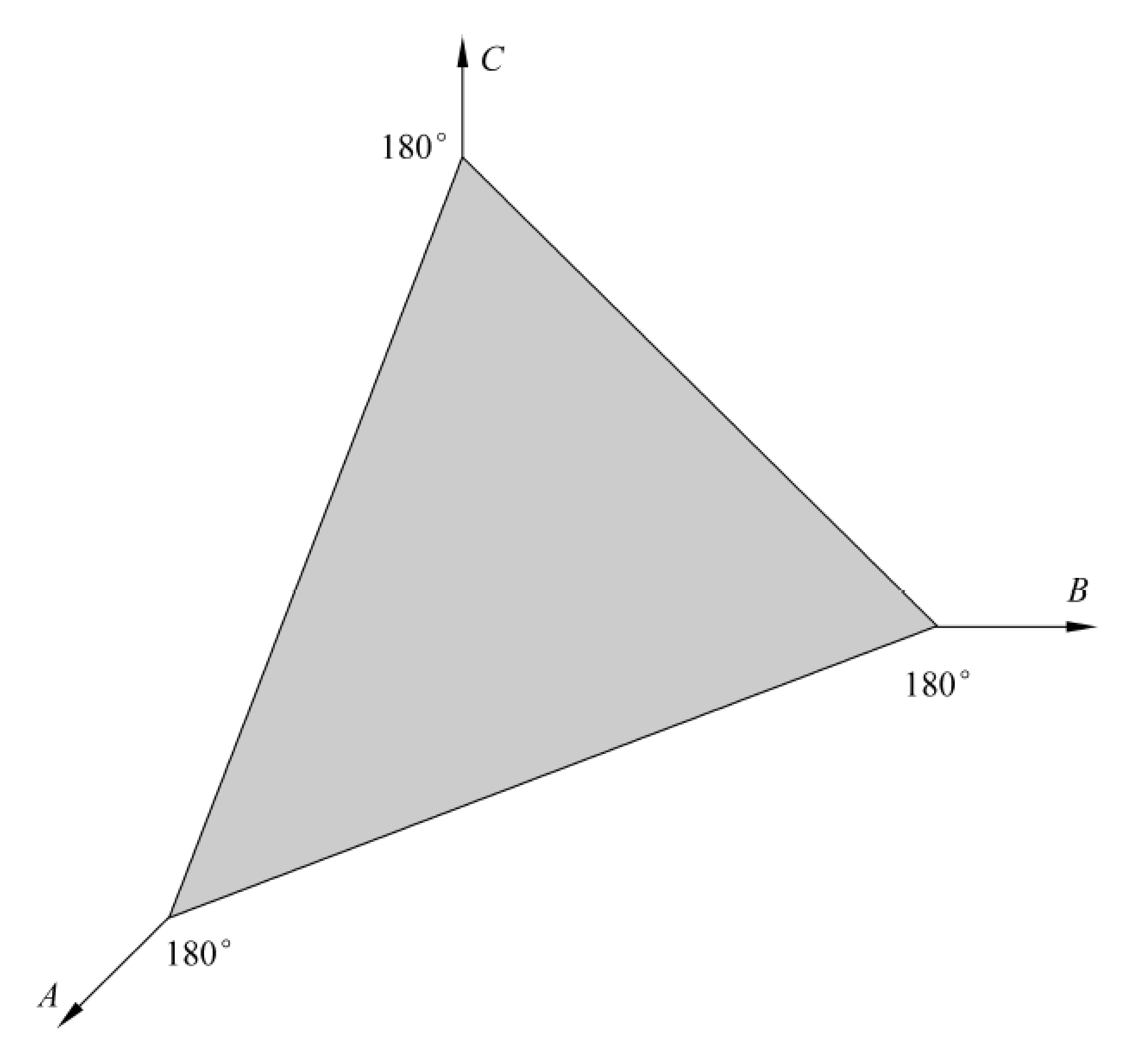


图 2-14 三角形角应满足的关系

一个角是钝角，即大于90°；直角三角形对应的输入条件是∠A、∠B 和∠C 中间有一个角是直角，即等于90°；锐角三角形对应的输入条件是∠A、∠B 和∠C 的角都为锐角，即都小于90°；非三角形对应的输入条件是 $A+B+C\neq180$。根据等价类的边界可以将整个输入域进行重新划分，如图 2-15 所示。整个输入域被分为 4 个小的三角形，其中最中间的小三角形表示锐角三角形，外面三个小的三角形表示钝角三角形，中间小三角形的边界表示直角三角形。

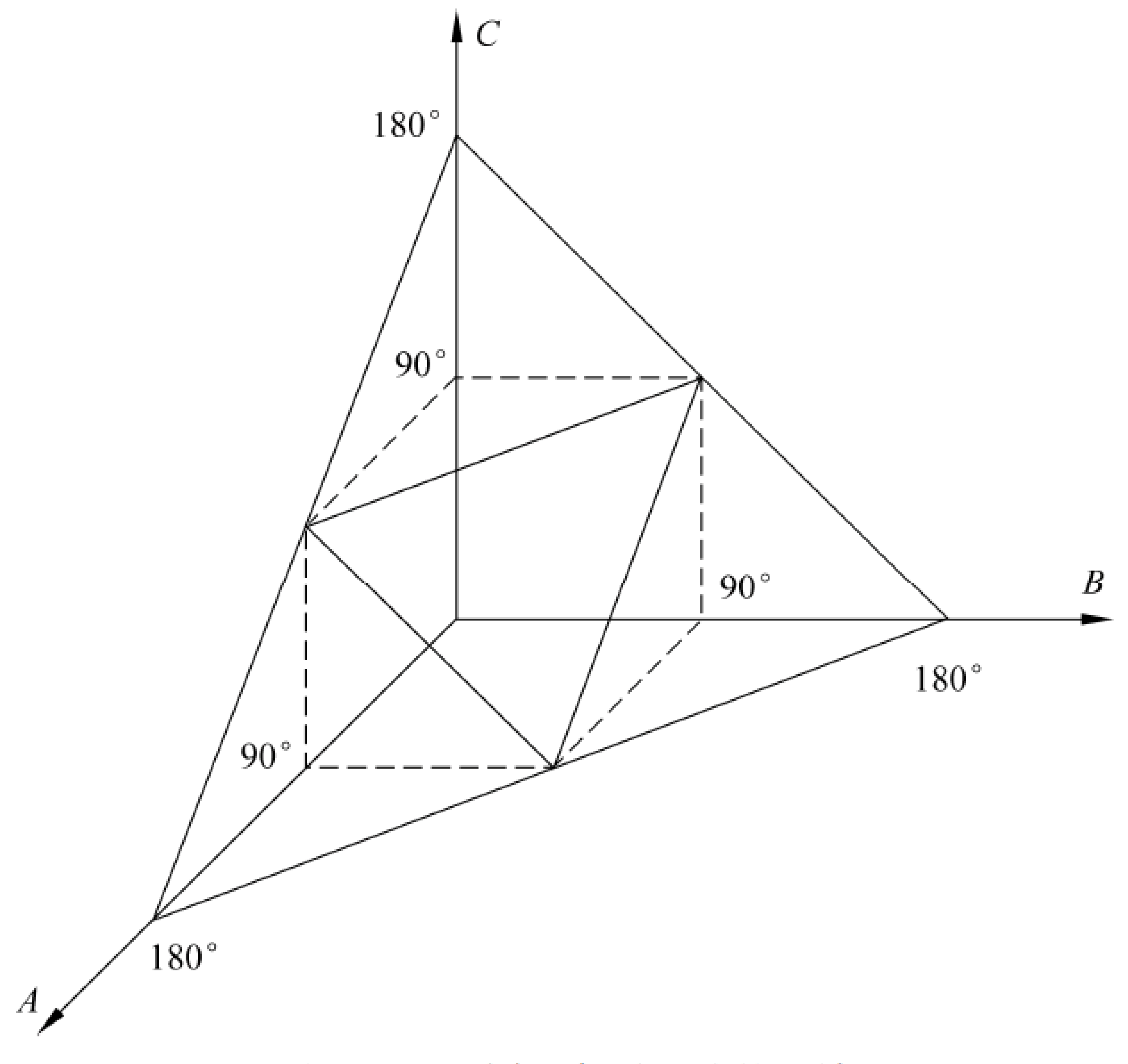


图 2-15 不同三角形所处的区域

设计测试用例，首先得知 A 的取值区间为(0，180)，测试用例中的 A 可取 0、45、90、135 和 180。如图 2-16 所示，表示 A 取不同的值对应的输入域上的直线。当 A＝180 时，是边界上的一个点，只需分析边界点周围的情况即可。当 A＝135 时，可以从图中看到，虚线与两个边界相交，所以设计测试用例时要考虑边界外的点、边界上的点以及边界内的点，总共 5 个点。当 A＝45 时，虚线与 4 个边界相交，设计测试用例时要考虑边界外的点、边界上的点以及边界内的点，总共 9 个点。当 A＝90 和 A＝0 时，正好是边界，则需要分析边界上特殊的点，一般是指边界相交的点。除此之外，这些点的附近也要分析。

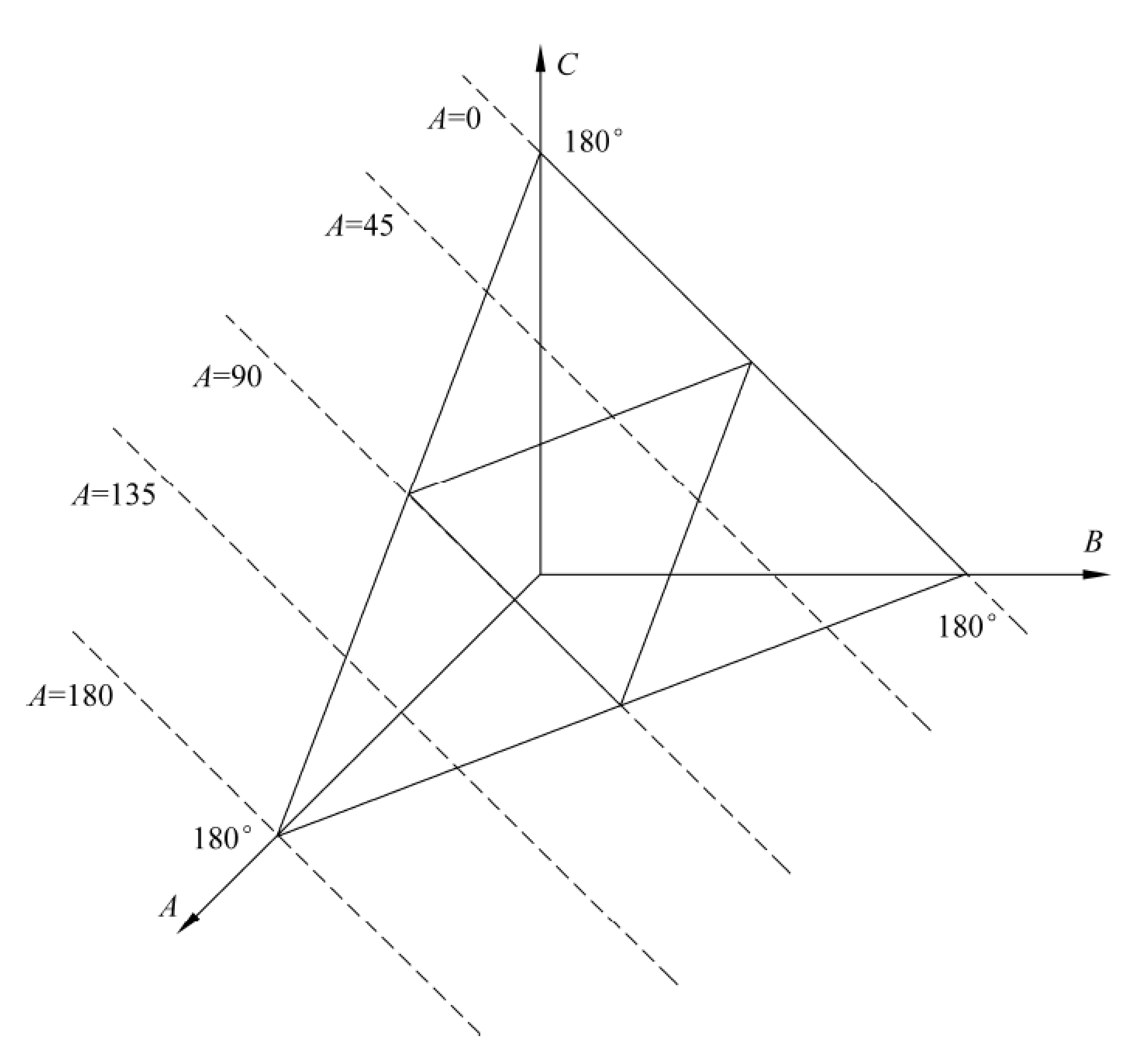


图 2-16　不同三角形边界取值分析

具体的用例设计如表 2-9 所示。

表 2-9　三角形问题的测试用例

用例编号	A	B	C	预期输出
1	0	0	179	不是三角形
2	0	0	180	不是三角形
3	0	1	180	不是三角形
4	0	90	89	不是三角形
5	0	90	90	不是三角形
6	0	91	90	不是三角形
7	0	180	−1	不是三角形

续表

用例编号	A	B	C	预期输出
8	0	180	0	不是三角形
9	0	181	0	不是三角形
10	45	0	136	不是三角形
11	45	0	135	不是三角形
12	45	5	130	钝角三角形
13	45	45	90	直角三角形
14	45	60	75	锐角三角形
15	45	90	45	直角三角形
16	45	130	5	钝角三角形
17	45	135	0	不是三角形
18	45	136	0	不是三角形
19	90	−1	90	不是三角形
20	90	0	90	不是三角形
21	90	0	91	不是三角形
22	90	45	45	直角三角形
23	90	90	−1	不是三角形
24	90	90	0	不是三角形
25	90	91	0	不是三角形
26	135	−1	45	不是三角形
27	135	0	45	不是三角形
28	135	5	35	钝角三角形
29	135	45	0	不是三角形
30	135	46	0	不是三角形
31	180	−1	0	不是三角形
32	180	0	0	不是三角形
33	180	0	1	不是三角形

2.2 等价类

使用所有可能数据测试程序是不大可能实现的，无法实现穷举测试。在测试过程中可从所有可能的输入中找出某个适当的子集进行测试，但是从测试效果上可以达到可以

接受的测试能力。通常采用划分等价类的方法选择合适的子集,以发现尽可能多的错误。划分等价类将所有可能的输入划分成若干个等价类,每个等价类中的一个典型值与该类中所有其他值所发现错误的能力是相同的。

等价类测试也属于黑盒测试,测试人员在使用等价类方法进行测试时只需按照需求规格说明设计测试用例,并根据测试用例的要求运行相关的测试方法,无须考虑程序的内部结构。

2.2.1 等价类的概念

在测试过程中,通过穷举一般很难覆盖所有测试的可能。因此,等价类测试过程中只取一组代表数据作为测试输入。

将大量的输入数据(有效的和无效的)划分为若干等价类时,等价类覆盖的范围通常是指输入域的子集合,在该集合中的所有数据对于发现错误的作用是等效的。等价类测试通常基于这样的假定:等价类中的某一代表值等价于这个等价类中所有其他值的测试,等价的含义是发现缺陷的能力是一样的。当利用某个等价类中的输入值进行测试时发现了错误,那么使用该等价类中所有输入值进行测试也会发现同样的错误;反之,如果使用等价类中的值无法发现错误,那么该等价类中的其他数据也无法发现错误。等价类发现缺陷的能力实际上是根据业务规则来分析其可能的发现缺陷能力。但是实际上,可能存在和程序实现相关联的缺陷,但是这些错误无法通过等价类来发现。

例如,对学生的考试成绩进行评级,其中评级的标准如表2-10所示。

表 2-10 学生成绩评级参考

成绩范围	级别	成绩范围	级别
90～100	优	60～69	差
80～89	良	<60	不及格
70～79	中		

当某个学生输入成绩时,学生系统能够自动判断出该生成绩的级别。当输入91、92、95时,判定为优;当输入81、83时,判定为良;当输入72、78时,判定为中;当输入66、68时,判定为差;当输入10、59时,判定为不及格。因此,(91,92,95)、(81,83)、(72,78)、(66,68)、(10,59)均可以被看作等价类。

等价类包括:有效等价类和无效等价类。

有效等价类是合理的、有意义的数据所构成的集合。利用有效等价类可以检验程序是否能够实现规格说明中规定的功能和性能。

无效等价类是不合理的、没有意义的数据构成的集合。测试人员主要利用无效等价类检验程序是否正确处理不符合规格说明要求的输入。

等价类划分是将程序的输入域划分为若干部分,然后从每个部分中选取少数代表性数据作为测试用例。每类的代表性数据在测试中的作用等价于这一类中的其他值。使用

等价类划分法设计测试用例通常有以下三步。

第一步：划分等价类；并对每一个等价类规定一个唯一的编号。

第二步：设计测试用例，使其尽可能多地覆盖尚未覆盖的有效等价类；重复该步骤直至所有的有效等价类均被覆盖。

第三步：设计新的测试用例，直至所有的无效等价类均被覆盖。

边界值分析法与等价类划分法都是黑盒测试中常用的方法，而边界值分析法是在等价类划分法的基础上进行的测试。这两种方法有以下两个不同之处。

（1）等价类划分法是在已分好的等价类中随机选取输入数据作为该等价类的代表，而边界值分析法是在已分好的等价类中选取边界值进行测试；

（2）等价属性不一定出现在边界上，但边界是一种等价属性表现。

2.2.2 等价类的划分及依据

等价类测试中最重要的步骤是等价类的划分，根据不同的输入数据的特征，可以设计不同的等价类。

原则 1：如果输入条件规定了取值范围、取值个数，则可以确立一个或者多个有效等价类或者无效等价类。

例如，在程序规格说明中给出的输入条件为："输入值为 0～100 之间的整数"，则有效等价类为"0≤输入值≤100"，无效等价类为"输入值＜0"或者"输入值＞100"。

原则 2：如果输入条件规定了输入值的集合，或者是规定了"必须/一定"的条件，可以确立一个有效等价类和一个无效等价类。

例如，在设置密码时规定密码必须不能为纯数字，则所有由数字构成的密码组成无效等价类，其他的就可以组成有效等价类。

原则 3：如果输入数据为一组限定值，同时需要对输入的值进行处理，此时可以为所有输入值确定一个有效等价类。此外，针对该值可以确立一个无效等价类，即所有不允许的输入值集合。

例如，为计算出差补贴，出差的住宿、交通工具以及餐费都给予报销。住宿宾馆按照 3 星级标准，交通工具限制为汽车、火车、飞机和轮船，餐费的报销也按照相应的标准。因此可以确定的有效等价类包括 3 星及 3 星以下的宾馆，汽车、火车、飞机和轮船，相应的伙食补助标准。除此以外，不符合以上条件的均可以被划分为无效等价类，如 4 星级宾馆、三轮车等的输入值集合。

原则 4：如果规定了输入数据必须遵守的规则，那么可以建立符合规则的有效等价类和其他若干无效等价类。

例如，Java 中的变量命名必须以 $ 、字母、下划线开头。这时的有效等价类可以是以 $ 、字母、下划线开头的名称，若干个无效等价类则是不是以 $ 、字母、下划线开头的名称。

原则 5：如果明确知道已划分的等价类中不同元素在程序中的处理方式不一样，那么将该等价类进一步划分为更小的等价类。

除了上述原则外，等价类划分还需要依据输入数据的类型、输入数据的层次关系、输

入数据的维度和输出条件等信息做进一步考虑。

依据一：根据输入数据的类型来划分等价类，主要分为两种情况：连续型和离散型。

1. 连续型

连续型是指输入数据属于连续型数据。输入数据有一个连续的取值区间，在这个区间上可以取测试数据。对于这种情况，首先要分析连续型数据取值区间上不同取值所代表的意义。根据不同的取值意义，找到相区别的边界值，再进行等价类划分。

例如，在一个处理水状态的程序中，在不同的温度下具有不同的属性，程序需要根据不同的状态执行不同的操作。0℃以下的水呈固态，0～100℃之间的水呈液态，100℃以上的水呈气态。对于输入的水温而言，温度值是连续型变量，其中，0℃和100℃是等价类的边界值，如图2-17所示。其中，(－6.5，－3)、(13，46)、(110，118)是等价类固态、液态和气态的数据。

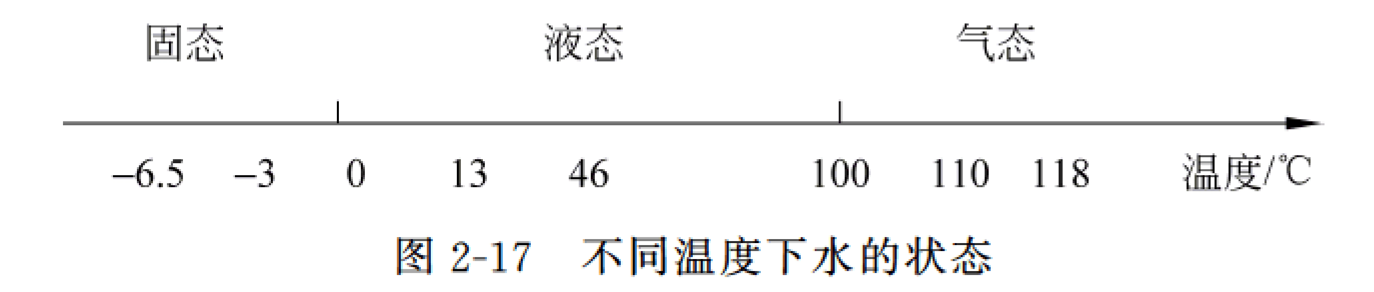


图2-17　不同温度下水的状态

2. 离散型

离散型是指输入数据属于离散型数据。输入数据的范围并不是一个连续的区间，而是离散地分布在区域中。离散的输入数据之间也是有关联的，一般方法是分析输入数据的属性，然后根据输入数据的一个或者多个属性对离散的数据进行分类，会得到不同的等价类。

例如，在一个某高校的教学管理系统中，若需要依据老师的职称以及所属的学院信息做不同处理，那么职称和所属院系可以作为等价类划分的依据。依据职称可将老师分为教授、副教授、讲师等。其中，(张三，李四，王五)、(赵一，钱二，孙三)是离散等价类数据，如图2-18(a)所示。如果依据所属的院系来分，可将老师分为信息学院、化工学院、商学院等。在职称属性中，张三、李四和王五属于一个等价类，而在所属院系属性中赵一、李四和孙三属于一个等价类，如图2-18(b)所示。离散的输入数据根据不同属性有不同的等价类划分。如果进行属性叠加分类，则等价类分得更细。

依据二：根据输入数据的层次关系来划分等价类，可以得到多层子等价类。

实际测试的时候，先从大范围来划分等价类，可以快速得到大的等价类。如果没有达到测试的粒度要求，需要对已有的等价类进一步划分，得到子等价类。子等价类的划分也是有必要的，有些情况下的等价类没有反映出其内部的差别，导致测试不全面。

例如，在教学管理系统中，教授根据其级别可以做进一步的划分，例如二级教授、三级教授、四级教授等。划分的粒度如何，依据业务处理是否区分这些信息。

依据三：根据输入数据的维度来划分等价类，主要分为两种情况：单维和多维。

单维：输入数据只有一个，只需要分析这一个输入数据的特性，并以此分析输入数据

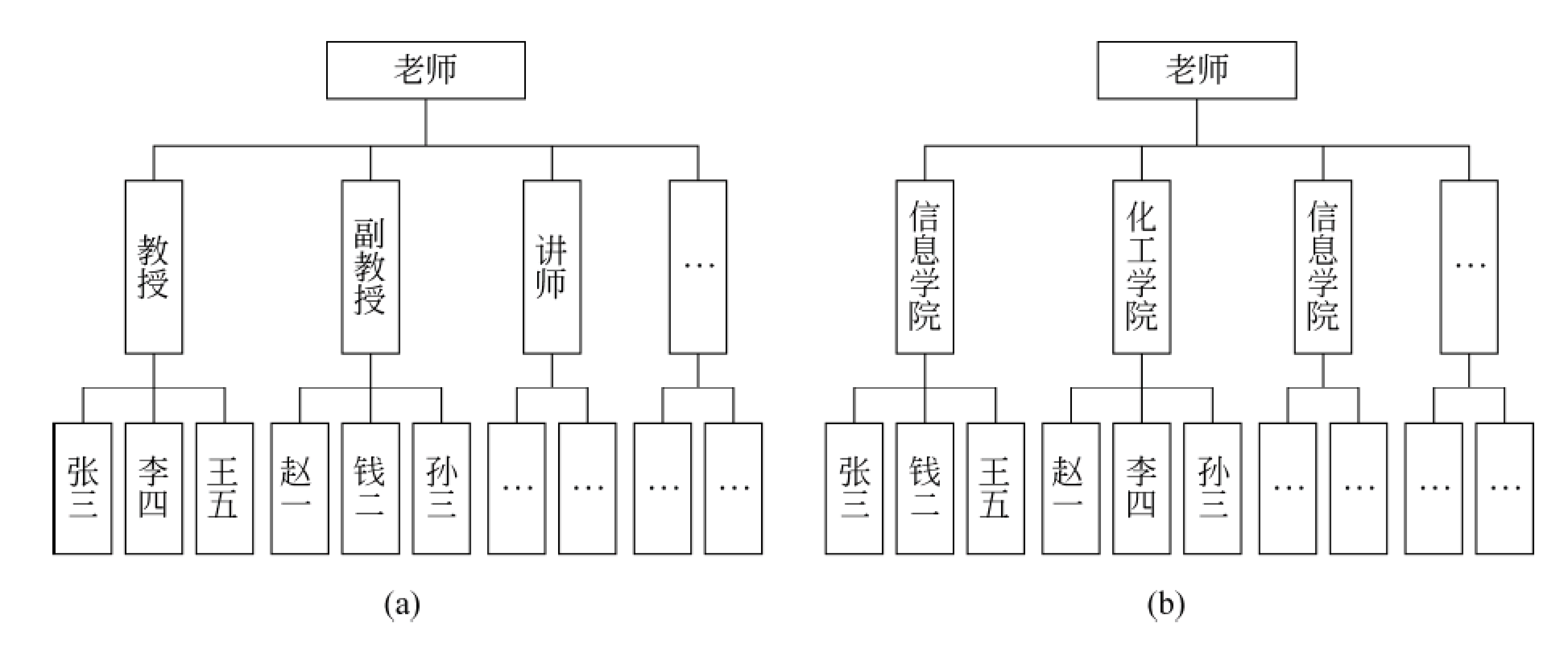


图 2-18　不同的数据属性可以划分不同的等价类

的连续性，然后划分等价类。

多维：不同于单维的输入数据，多维涉及多个数据变量，就会存在多种组合。其中一种情况是各维度之间没有必然的关系，则应先对每一个输入变量各自划分等价类，然后将所有的维度相互组合形成新的多维等价类。另一种情况是各维度之间是有一定关联的，各个变量的组合是表示一定意义的(有些组合可能没有意义)，则根据组合的意义来划分等价类，而不是单纯的相互组合。

例如，用户注册信息中需要填写用户名和密码。对用户名的要求是未注册过的，则可以将用户名分为已注册的和未注册的。对密码的长度要求是 6 位以上 20 位以下，则可以将密码分为长度在 6 位以下、长度在 6 位以上 20 位以下，以及长度在 20 位以上的等价类。如果同时考虑用户名和密码的关系，那么可以构成 6 个测试用例。

例如，用户登录信息中需要填写用户名和密码。登录成功的要求是用户名存在且对应的密码输入正确。可以看到这种情况下的用户名和密码是关联的。所以测试等价类可以首先分为两大类：登录成功和登录失败。其中，登录失败又可以分为用户名不存在和用户名存在但是密码错误两类。这种情况的等价类划分考虑的是输入变量间的组合关系，而不是简单的相互组合。例如，用户名不存在和密码正确这种组合是没有意义的，不可能存在。

依据四：根据输出条件来划分等价类。

规格说明书中并不一定对输入条件进行约束说明，很多情况下是对输出条件进行说明。不同的输出对应不同的等价类，所以输出对输入也有一定的约束。

上面提供的几种等价类划分依据相互之间并不冲突，在实际测试时可以考虑多个方面。

2.2.3　等价类测试举例

以 NOIP1182 食物链问题为例，来说明等价类测试的方法。具体的问题描述如下。

动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B，B 吃 C，C 吃 A。现有 N 个动物，以 $1-N$ 编号。每个动物都是 A,B,C 中的一种，但是并不

知道它到底是哪一种。用两种说法对这 N 个动物所构成的食物链关系进行描述：

第一种说法是“1 X Y”，表示 X 和 Y 是同类。

第二种说法是“2 X Y”，表示 X 吃 Y。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

(1) 当前的话与前面的真话冲突，就是假话。

(2) 当前的话中 X 或 Y 比 N 大，就是假话。

(3) 当前的话表示 X 吃 X，就是假话。

根据给定的 $N(1\leqslant N\leqslant 50\ 000)$ 和 K 句话 $(0\leqslant K\leqslant 100\ 000)$，输出假话的总数。

由上面的描述可知，程序的输入是动物的数量 N 和 K 句话，由于每一句话包含两个动物，分别用 X 和 Y 标示，那么实际参数是 N、X 和 Y。实际测试时，输入的一句话可以进行等价类划分，首先可以分为真话和假话两大类，不存在又不是真话又不是假话的情况。其中假话可以继续划分成三个子等价类，分别是：

(1) 当前的话与前面的真话冲突；

(2) 当前的话中 X 或 Y 比 N 大；

(3) 当前的话表示 X 吃 X。

真话的要求其实就是无法满足假话的任一条件，即必须同时满足当前的话与前面的某些真话不冲突、当前的话中 X 和 Y 都比 N 小以及当前的话表示的不是 X 吃 X。真话也可以划分为两类，分别是用第一种说法和用第二种说法。具体的等价类划分如图 2-19 所示，大的等价类是真话和假话，真话又根据说的话类型分为两个子等价类，同样假话根据假话的条件分为三个子等价类。

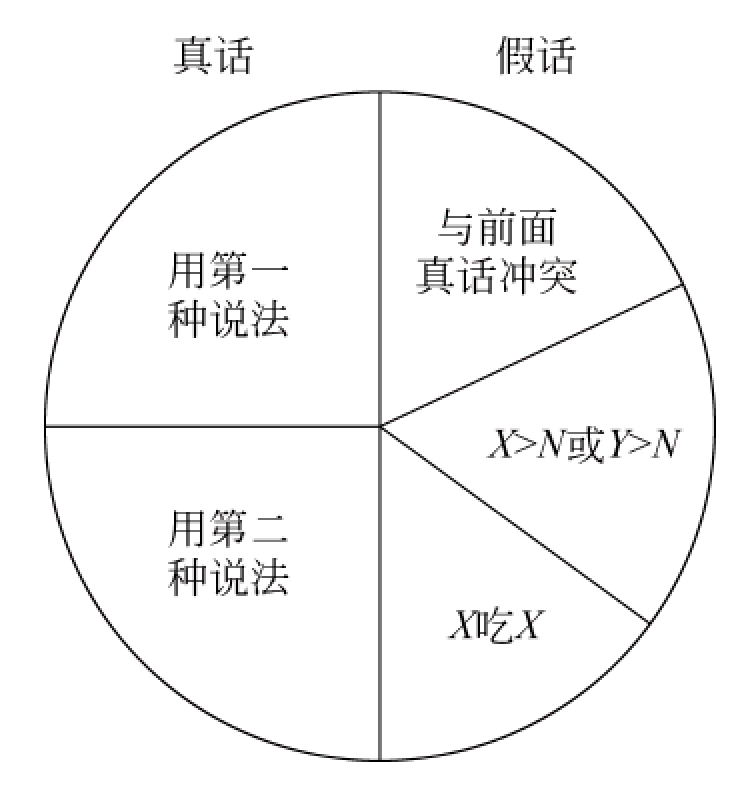


图 2-19　第一个维度的等价类划分

如果现在 $N=5$，$K=4$，并且已经输入了三句话，分别是“2 1 3”(1 吃 3)、“2 3 2”(3 吃 2)以及“2 2 1”(2 吃 1)。最后的输出结果是假话的个数，因为前面三句都是真话，所以假话的个数是 1 就代表最后一句话是假，否则为真。根据上述两种等价类划分方式可以设计不同的测试用例来测试这个算法，第一种划分方式的测试用例设计如表 2-11 所示。注意，这里每一行用例之间没有依赖关系，都是单独和本段开始描述的三句话一起构成连续的 4 句话。

表 2-11　第一种划分方式的测试用例设计

用例编号	测 试 用 例	预期输出(1 表示假话)
1	1 1 4(1 和 4 是同类)	0
2	2 5 1(5 吃 1)	0

续表

用例编号	测 试 用 例	预期输出(1 表示假话)
3	2 3 1(3 吃 1)	1
4	1 6 1(6 和 1 是同类)	1
5	2 3 3(3 吃 3)	1

除了上面的分类,还可以有第二种分类方式。首先把话分为用第一种说法和用第二种说法两大类。每一类又可以分为真话和假话两类。假话可以再进一步划分,但是不像上一种划分方式,都根据假话的三个条件分为三类。用不同种说法说的假话划分子类情况不同:用第二种说法说的假话仍然可根据假话的三个条件分为三类;用第一种说法说的假话可分为两类,分别满足前面两个条件,第三个条件不存在,因为第一种说法 X 和 Y 是同类,不存在 X 吃 X 的情况,则不需要再划分这个子类。具体的等价类划分如图 2-20 所示。

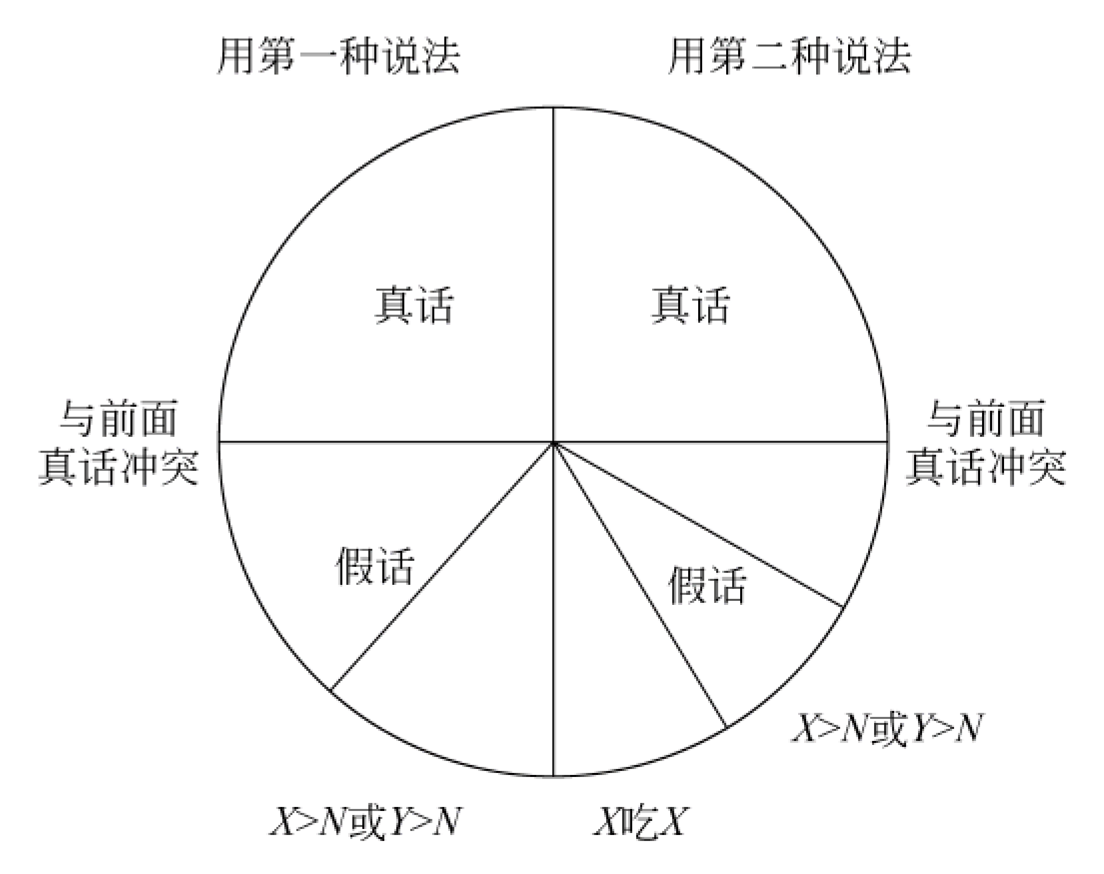


图 2-20　第二个维度的等价类划分

根据第二种划分等价类的方法,可以设计相关的测试用例,如表 2-12 所示。

表 2-12　第二种划分方式的测试用例设计

用例编号	测 试 用 例	预期输出(1 表示假话)
1	1 1 4(1 和 4 是同类)	0
2	1 2 3(2 和 3 是同类)	1
3	1 6 1(6 和 1 是同类)	1
4	2 5 1(5 吃 1)	0
5	2 3 1(3 吃 1)	1
6	2 6 1(6 吃 1)	1
7	2 3 3(3 吃 3)	1

比较上面两种方式，可以发现：第一种划分方式中的假话可分为三类，而第二种划分方式中用第一种说法的假话只分为两类。其实是假话的大类对假话的划分有着一定的约束，所以导致同样属性的等价类分的子类情况不同。等价类划分中的子类划分可根据实际情况需要进行划分。一般来说，等价类分得越细，则覆盖的测试用例越全面。

2.3 决策表

2.3.1 决策表的概念

决策表也被称为判定表(Decision Table)，适合描述在不同逻辑条件取值组合的情况下需要执行的动作。决策表通常由 4 个部分组成，如图 2-21 所示。

(1) 条件桩(Condition Stub)：列出问题中可能出现的条件，一般情况下条件的次序是无关紧要的。

(2) 动作桩(Action Stub)：列出解决问题可能采取的操作，一般情况下操作的排列顺序没有约束。

(3) 条件项(Condition Entry)：针对所有条件的取值列出不同条件取值的组合。

(4) 动作项(Action Entry)：在条件项各种取值的情况下应该采取的动作。

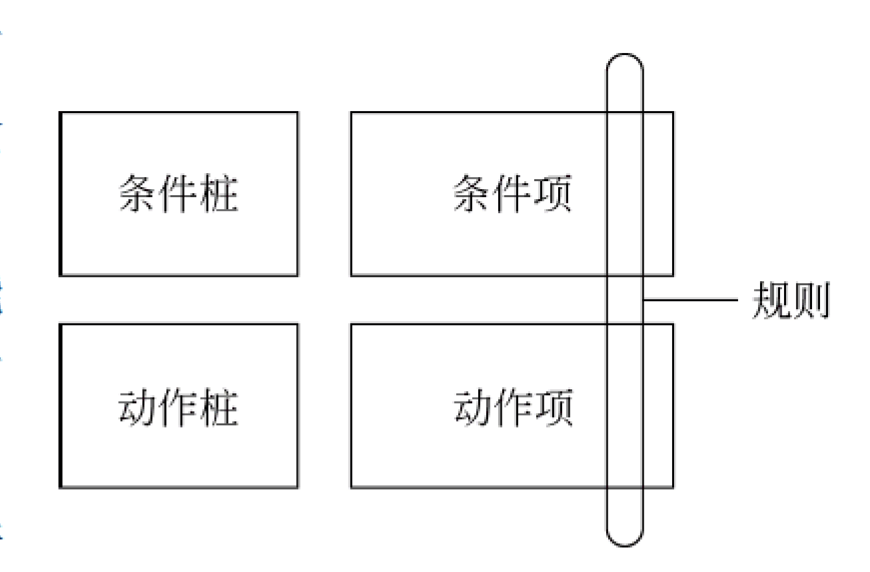


图 2-21　决策表结构

通常情况下，决策表对于每个条件的先后次序是没有要求的，除了某些特定问题有特殊要求。为便于阅读可以人为指定所有动作的排列顺序，一般采用相应的“规则”进行操作。所谓的“规则”就是任何一个条件组合以及相关的执行操作，在决策表中对应为纵向贯穿条件项和动作项的一列。因此，决策表中所列出的条件取值组合就表示有多少规则。

2.3.2 决策表的建立

通常根据软件规格说明，按照以下步骤建立决策表。

(1) 确定规则数目。如果有 n 个条件，每个条件的取值为(0,1)两种，便有 2^n 种规则。

(2) 列出所有的条件桩和动作桩。

(3) 输入条件项。

(4) 输入动作项，制定初始决策表。

(5) 简化，合并相似或者相同的动作。

Beizer 指出了适合使用决策表设计测试用例的条件。

(1) 规格说明以决策表的形式给出或者很容易转换成决策表。

(2) 输入条件的排列顺序不影响操作执行。

(3) 规则的排列顺序不影响操作执行。

(4) 当某一规则的条件已满足,同时确定需要执行的操作时,无须检验其他规则。

(5) 如果某一规则需要执行多个操作,这些操作的顺序不会造成影响。

决策表结构如表 2-13 所示。

表 2-13 决策表的结构

选项 \ 规则		规则 1	规则 2	规则 3、4	规则 5、6	规则 7	规则 8
条件	c1	T	T	T	F	F	F
	c2	T	T	F	—	T	F
	c3	T	F	—	T	F	F
动作	a1	Y					
	a2		Y	Y			
	a3				Y		Y
	a4					Y	

表中的 c1、c2 和 c3 表示条件桩,a1、a2、a3 和 a4 表示动作桩,决策表的每一列对应一条规则,每条规则包括一个条件组合(条件项)以及相关的执行操作(动作项)。每个条件的取值为(0,1)两种,用 T 和 F 表示。如果有 n 个条件,便有 2^n 条规则。

决策表适用于很多情况,它可以将问题细化,将复杂的问题分解成多种情况。分解后的问题更容易理解,并且可以直接通过决策表设计测试用例,不容易遗漏情况。决策表适用于以下的情况。

(1) 输入变量和输出变量之间有因果关系。

(2) 输入变量之间存在着一定的逻辑关系。

(3) 输入域可以进行分解,且对应不同的输出。

决策表不适合有执行顺序的动作或者重复的动作,在使用决策表时要考虑是否适用。

2.3.3 决策表的简化

建立决策表后,部分决策表可以进行简化,决策表的简化包括以下三个方面。

1. 合并相似项

当两个条件项对应相同的动作,且只有一个条件的值是不同的,则可以将这两个条件项进行合并,合并后的条件项需要将不同值的条件设为不关心,用—表示,如图 2-22 所示。不同值的条件对结果不影响,被称为无关项或者不关心条目,用—或 N/A 表示。

2. 合并包含项

如果一个条件项已经包括另一个条件项,执行的动作相同则也可以进行合并,如

图 2-23 所示。一般情况下，当有无关项出现时，则需要注意是否有其他的条件项已经被其包含，如果包含，则保留无关项，删除包含项。

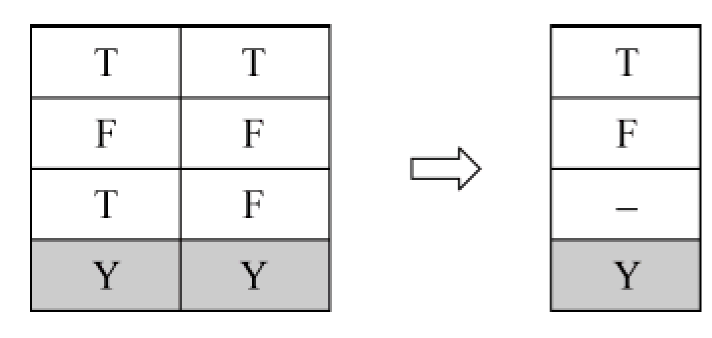



图 2-22 合并相似项

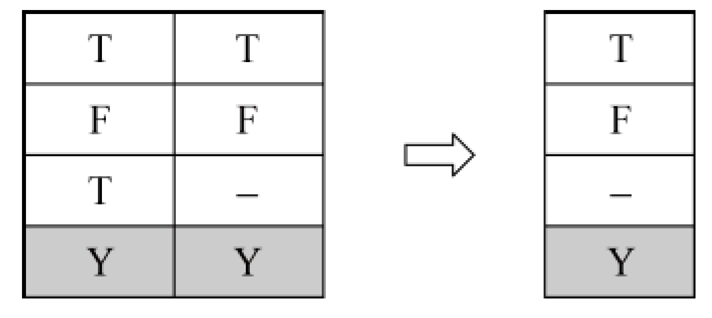


图 2-23 合并包含项

3. 合并不可能项

如果决策表中有相互排斥条件，则当一个选项为真时，其他几个条件就一定是假的。简化时将其他条件用—表示。这里的不关心条目表示“必须失败”，而不是随意取值。例如，c1、c2 和 c3 是三个互斥的条件，则决策表如表 2-14 所示。规则 1 的含义是当 c1 为真时，c2 和 c3 一定是假的。在设计测试用例时要注意—的具体含义。

表 2-14 合并不可能项

选项 \ 规则		规则 1	规则 2	规则 3
条件	c1	T	—	—
	c2	—	T	—
	c3	—	—	T
动作	a1			
	a2			
	a3			

合并条件项的过程其实就是对决策表进行简化，方便测试用例的设计。下面举一个例子说明规则的合并过程。

现有一台 ATM 自动取款机，插入银行卡后必须输入正确的密码才能取款。如果账户内的余额不足会提示卡内余额不足，如果 ATM 内现金不足会提示 ATM 内现金不足。具体的决策表如表 2-15 所示。

表 2-15 ATM 决策表

选项 \ 规则		1	2	3	4	5	6	7	8
条件	c1：密码是否正确	T	T	T	T	F	F	F	F
	c2：账户内余额是否足够	T	T	F	F	T	T	F	F
	c3：ATM 内现金是否足够	T	F	—	F	T	F	T	F

续表

选项 \ 规则		1	2	3	4	5	6	7	8
动作	a1：取款成功	Y							
	a2：提示密码错误					Y	Y	Y	Y
	a3：提示卡内余额不足			Y	Y				
	a4：提示 ATM 内现金不足		Y						

分析表 2-15，可以发现条件项 3 包含条件项 4，则可以直接删除规则 4。条件项 5 和 6 是相似项，对应相同的动作，只有 c3 不同，则可以将这两个条件项合并。同理，条件项 6 和 7 是相似项，也可以进行合并。这里还发现一个有趣的现象，条件项 5 和条件项 7 也是相似项，条件项 6 和条件项 8 也是相似项，将这 4 个条件项合并，得到最终的决策表如表 2-16 所示。

表 2-16 化简后的 ATM 决策表

选项 \ 规则		1	2	3，4	5—8
条件	c1：密码是否正确	T	T	T	F
	c2：账户内余额是否足够	T	T	F	—
	c3：ATM 内现金是否足够	T	F	—	—
动作	a1：取款成功	Y			
	a2：提示密码错误				Y
	a3：提示卡内余额不足			Y	
	a4：提示 ATM 内现金不足		Y		

2.3.4 决策表规则数统计

完备决策表（所有规则都列出来）的规则条目就是决策表中列的数目，即一列表示一条规则。但实际分析中，经常会将决策表进行简化，就会有无关项，从而影响规则条数统计。下面将进行具体分析。

原则一：对于没有互相排斥条件的决策表，如果有 n 个条件，则产生 2^n 条规则。如果决策表中没有无关项，每一列对应一个规则；如果决策表中有无关项，则每出现一个无关项，该列的规则数目乘以 2。例如，表 2-15 是完备决策表，规则数目是 2^3，决策表的列数也是 8 条。表 2-16 是简化后的决策表，第一列和第二列各表示一条规则；第三列有一个无关项，则表示 $1\times2=2$ 条规则；第四列有两个无关项，则表示 $1\times2\times2=4$ 条规则。

原则二：对于有互相排斥条件的决策表，如果决策表中没有无关项，则和原则一一样，每一列对应一个规则；如果决策表中有无关项，则需要分析无关项的含义。如果无关项的含义是条件无关，则仍然是每出现一个无关项，该列的规则数目乘以2。如果无关项的含义是“必须失败”，需要扩展决策表使决策表完备，然后再计算规则数，不能直接计算。

例如，表2-17显示的是含有互相排斥条件的决策表，如果直接按照原则一，决策表中增加了规则条数统计这一行，最后得到的规则条数是4+4+4=12。

表2-17　含有互斥条件的决策表

选项 \ 规则		规则1	规则2	规则3
条件	c1	T	—	—
	c2	—	T	—
	c3	—	—	T
规则条数统计		4	4	4
动作	a1			
	a2			
	a3			

这12个规则中是有重复的规则，同时又没有覆盖所有的规则，如表2-18所示。

表2-18　将表2-17展开以后的决策表

选项 \ 规则		1	2	3	4	5	6	7	8	9	10	11	12
条件	c1	T	T	T	T	T	T	F	F	T	T	F	F
	c2	T	T	F	F	T	T	T	T	T	F	T	F
	c3	T	F	T	F	T	F	T	F	T	T	T	T
规则条数统计		1	1	1	1	1	1	1	1	1	1	1	1
动作	a1												
	a2												
	a3												

从表2-18中可以看到，规则1、规则5和规则9是重复的，规则2和规则6是重复的，规则3和规则10是重复的，规则7和规则11是重复的。去掉重复的规则，最后得到12−5=7条规则。实际上表中还缺少了所有条件都为假的规则，所以实际上的规则数是8条。所以，必须按照原则二来统计规则条数。需要扩展决策表到原始的完备表，删除重复的规则和增加缺少的规则，如表2-19所示。

表 2-19 实际的决策表

选项＼规则		1	2	3	4	5	6	7	8
条件	c1	T	T	T	T	F	F	F	F
	c2	T	T	F	F	T	T	F	F
	c3	T	F	T	F	T	F	T	F
规则条数统计		1	1	1	1	1	1	1	1
动作	a1								
	a2								
	a3								

2.3.5 决策表特性

决策表直接会影响到测试用例的设计，在建立决策表阶段需要关注决策表是否满足三个特性：完备性、无冗余性和一致性。

首先，完备性是指决策表中需要包括所有的规则。例如，表 2-20 表示的决策表没有满足完备性，缺少了全部为假的条件项，则需要增加这一条件项。通常情况下，不满足完备性的原因是设计者在设计测试表时会忽略一些情况，需要全面考虑。

表 2-20 不完备的决策表

选项＼规则		1	2	3	4	5
条件	c1	T	T	T	F	F
	c2	T	F	F	—	T
	c3	—	T	F	T	F
规则条数统计		1	1	1	1	1
动作	a1	Y		Y		
	a2		Y			Y
	a3				Y	

其次，无冗余性是指决策表中没有重复的规则。例如，表 2-21 中的条件项 1 包含条件项 2，而且它们对应的动作是一致的，决策表存在冗余。需要修改决策表，删除重复的条件项。直接冗余发生的比较少但也有可能发生，例如决策表中有完全相同的两列。发生冗余通常和无关项有关，无关项包含很多种情况，如果把包含的情况也列出来就会出现冗余。关注无关项的冗余是检查决策表是否冗余很重要的一点。

表 2-21　冗余决策表

选项＼规则		1	2	3	4	5
条件	c1	T	T	T	T	F
	c2	T	F	T	F	—
	c3	—	T	T	F	—
规则条数统计		1	1	1	1	1
动作	a1	Y	Y			
	a2			Y		Y
	a3				Y	

最后，一致性是指决策表中没有相互冲突的规则，表示两个规则的条件项相同或者一个条件项包含另一个条件项，但是执行的动作不同。表 2-22 中条件项 1 包含条件项 2，但是它们执行的动作不同，说明决策表不一致，需要分析实际情况再重新设计规则。

表 2-22　不一致的决策表

选项＼规则		1	2	3	4	5
条件	c1	T	T	T	T	F
	c2	T	F	T	F	—
	c3	—	T	T	F	—
规则条数统计		1	1	1	1	1
动作	a1	Y				
	a2		Y	Y		Y
	a3				Y	

2.3.6　决策表测试用例设计

本节以一个货运计费系统为例来说明决策表测试用例的设计。

货运收费标准如下：如果收货地点在本省以内，重量小于 15kg，快件 10 元/kg，慢件 5 元/kg；重量大于 15kg，超出部分每千克收费增加 3 元/kg。如果收货地点在本省以外，重量小于 15kg，快件 15 元/kg，慢件 10 元/kg；重量大于 15kg，超出部分每千克收费增加 3 元/kg(重量用字母 W 表示)。

根据以上描述分析得到，收费的条件有以下三个。

c1：是否在本省以内

c2：是否快件

c2：重量是否超出 15kg

将上述条件两两组合可以得到 8 种情况，但因为省内快件和省外慢件收费方式相同，实际得到的收费方式有以下 6 种。

a1：$5 \times W$

a2：$10 \times W$

a3：$5 \times W + 3 \times (W - 15)$

a4：$10 \times W + 3 \times (W - 15)$

a5：$15 \times W$

a6：$15 \times W + 3 \times (W - 15)$

根据条件和动作（收费）之间的关系可以得到如表 2-23 所示的决策表。

表 2-23　货运计费问题的决策表

	条　件	1	2	3	4	5	6	7	8
条件	c1：是否在本省以内？	T	T	T	T	F	F	F	F
	c2：是否快件？	F	F	T	T	F	F	T	T
	c2：重量是否超出 15kg？	F	T	F	T	F	T	F	T
收费	a1：$5 \times W$	Y							
	a2：$10 \times W$			Y		Y			
	a3：$5 \times W + 3 \times (W - 15)$		Y						
	a4：$10 \times W + 3 \times (W - 15)$				Y		Y		
	a5：$15 \times W$							Y	
	a6：$15 \times W + 3 \times (W - 15)$								Y

根据上述决策表可以设计一组测试用例，如表 2-24 所示。

表 2-24　货运问题的测试用例

用例编号	测 试 用 例	预期输出（收费）
1	省内慢件，10kg	10×5＝50
2	省内慢件，20kg	5×20＋3×（20－15）＝115
3	省内快件，10kg	10×10＝100
4	省内快件，20kg	10×20＋3×（20－15）＝215
5	省外慢件，10kg	10×10＝100
6	省外慢件，20kg	10×20＋3×（20－15）＝215
7	省外快件，10kg	15×10＝150
8	省外快件，20kg	15×20＋3×（20－15）＝315

2.4 因果图

2.4.1 因果图的概念

因果图，顾名思义，就是包含原因、结果以及它们之间关系的一种图。在因果图中，原因和原因之间、原因和结果之间都存在一定的关联。原因是指输入条件或者输入条件的等价类，结果是指输出条件。

在因果图设计测试用例的过程中，需要用一定的图形符号来表示一些约束和限制。表示原因和结果之间的关系有四 种，分别是恒等、非、或和与，具体如图 2-24 所示。左侧的节点表示原因(输入条件)的状态，用 C1 表示，右侧的节点表示结果(输出条件)的状态，用 E1 表示。用一条直线连接两个节点，并用相对应的逻辑符号来表示原因和结果之间的逻辑关系。节点状态可取 0 或者 1，其中 0 表示“不存在”状态，1 表示“存在”状态。

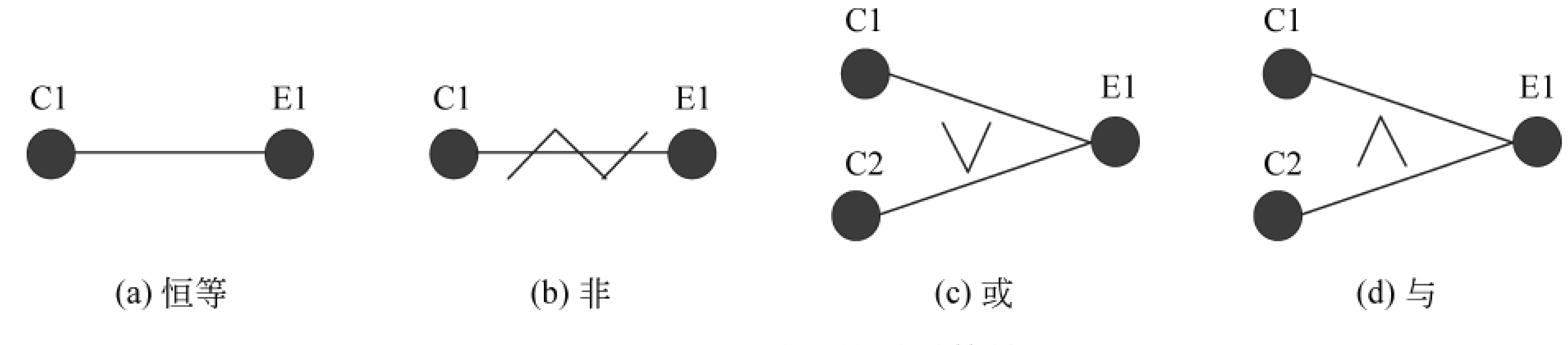


图 2-24 因果图的图形符号

(1) 恒等(identity)：表示原因和结果一一对应关系。如果 C1=0，则 E1=0；如果 C1=1，则 E1=1。

(2) 非(not)：表示原因和结果的相反关系。如果 C1=0，则 E1=1；如果 C1=1，则 E1=0。

(3) 或(or)：表示多个原因中只要有一个是存在状态，则结果就存在。如果 C1 或 C2 是 1，则 E1 是 1，否则 E1 是 0。

(4) 与(and)：表示多个原因都是存在状态，结果才存在。如果 C1 和 C2 都是 1，则 E1 是 1，否则 E1 是 0。

除了原因和结果之间存在联系，各原因之间以及各结果之间也会存在一些联系，这些联系称为“约束”。在因果图中，有相应的约束符号表示各种约束，例如互斥、包含、唯一、要求和屏蔽。具体的约束符号表示如图 2-25 所示。

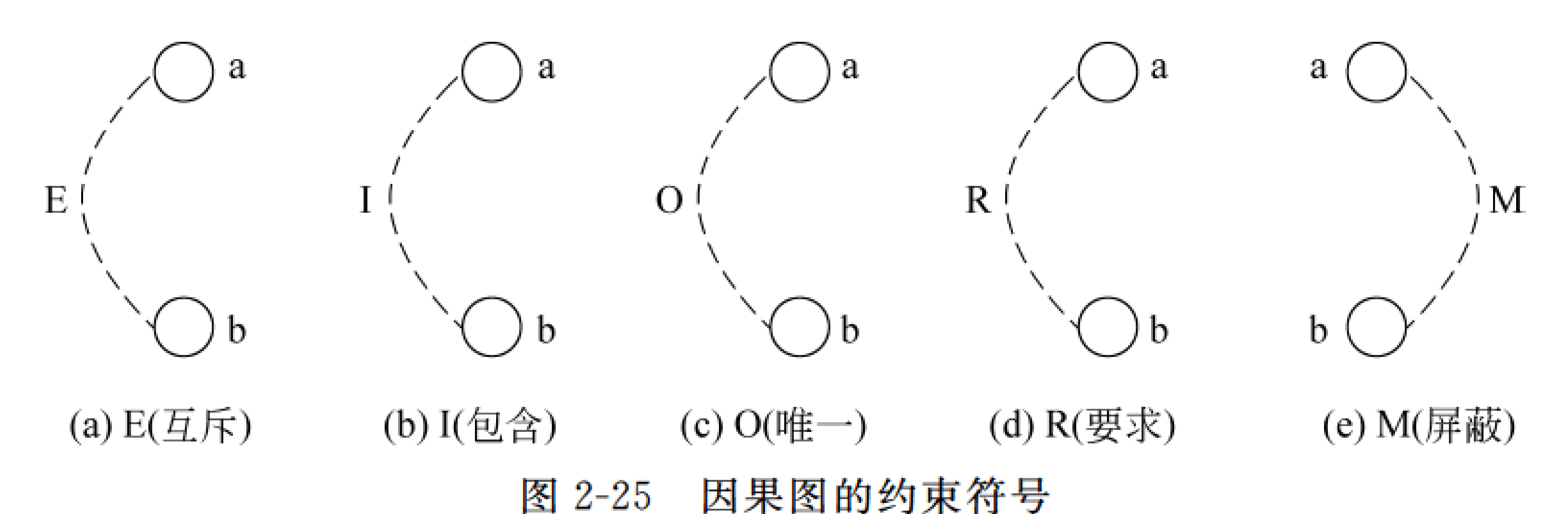


图 2-25 因果图的约束符号

(1) 异(E 约束)：表示多个原因中最多有一个是成立的。即 a 和 b 中最多有一个是 1,可能都是 0,但不可能都是 1。

(2) 或(I 约束)：表示多个原因中至少有一个是成立的。即 a 和 b 中至少有一个是 1,可能都是 1,但不可能都是 0。

(3) 唯一(O 约束)：表示多个原因中有且只有一个是成立的。即 a=1,b=0 或者 a=0,b=1,a 和 b 中有且只有一个为 1。

(4) 要求(R 约束)：表示多个原因同时成立或者同时不成立。即 a 和 b 同时为 0 或者 1。

(5) 强制(M 约束)：表示 a=1 时,b 强制为 0。其他情况下没有约束。

以上 5 个约束中,前面 4 个是对原因(输入条件)的约束,只有最后一个是对结果(输出条件)的约束。

2.4.2 因果图设计

利用因果图设计测试用例的具体步骤如下。

(1) 阅读软件规格说明书并进行分析,找到原因和结果,即输入条件和输出条件,用统一的标识符对原因和结果进行标识。

(2) 仔细分析软件规格说明书中的语义,确定各原因之间、各结果之间以及原因和结果之间的关系,并将这些关系转化成因果图。

(3) 因为原因和原因、结果和结果以及原因和结果之间有关联,需要在因果图中用一些图形符号来表示这些约束和限制。

(4) 分析因果图中的各种条件关系,将其转化为判定表。

(5) 根据判定表中的每一列数据设计一个测试用例。

2.4.3 利用因果图设计测试用例

以升降横移类立体车库为例,如图 2-26 所示。它是采用载车板升降或横移来存取车辆,每个立体车库有一个位置没有载车板(空位),是为了给载车板的移动提供移动空间。现在具体分析两层的升降横移类立体车库。由图 2-26 可知,两层的车位中至少一个是空



图 2-26 立体车库

位，用于其他载车板的移动。第一层停放的车辆可以直接取车，第二层的车辆需要下降到第一层再取车。注意，在下降载车板时要确保下面是空位。如果不是，则需要移动其他载车板来制造这种情况。停入车库的每辆车都有相应的编号，当需要取车时，会自动根据编号来获取车的位置和空位的位置并通过移动载车板将车从车库取出。

在不同位置上的车子移动的方向，如图 2-27 所示。

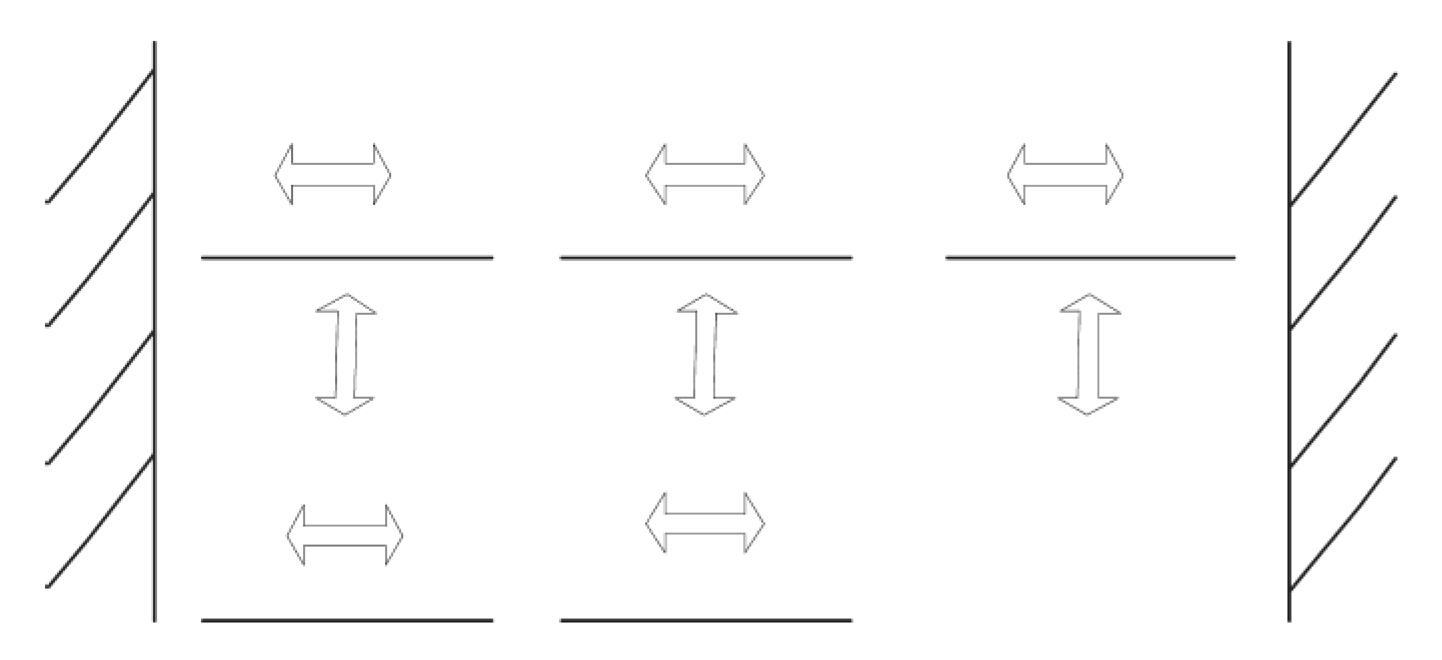


图 2-27　车子移动方向示意图

下面分析取车的过程原因和结果，记录产生的中间状态，如表 2-25 所示。

表 2-25　取车过程的原因和结果

原因	C1：取车位在第一层 C2：取车位在第二层 C3：空位在第一层 C4：空位在第二层 C5：空位在取车位正下方 C6：空位不在取车位正下方
中间状态	L1：开始时取车位在第一层且空位在第一层 L2：开始时取车位在第一层且空位在第二层 L3：开始时取车位在第二层且空位在取车位正下方 L4：开始时取车位在第二层且空位在第一层但不在取车位下方 L5：开始时取车位在第二层且空位在第二层 L6：取车位在第一层且空位在第二层 L7：取车位在第二层且空位在取车位正下方 L8：取车位在第二层且空位在第一层但不在取车位下方
结果	E1：直接取车

根据原因和结果的分析，可以得到因果图，如图 2-28 所示。

注意：从中间节点 L5 到 L8，需要上升空位下方载车板让出下层空位；从中间节点 L8 到 L7，需要横移其他载车板让出取车位下方空位；从中间节点 L7 到 L6，需要下降取车位到第一层。

根据因果图可以得到相应的决策表，如表 2-26 所示。

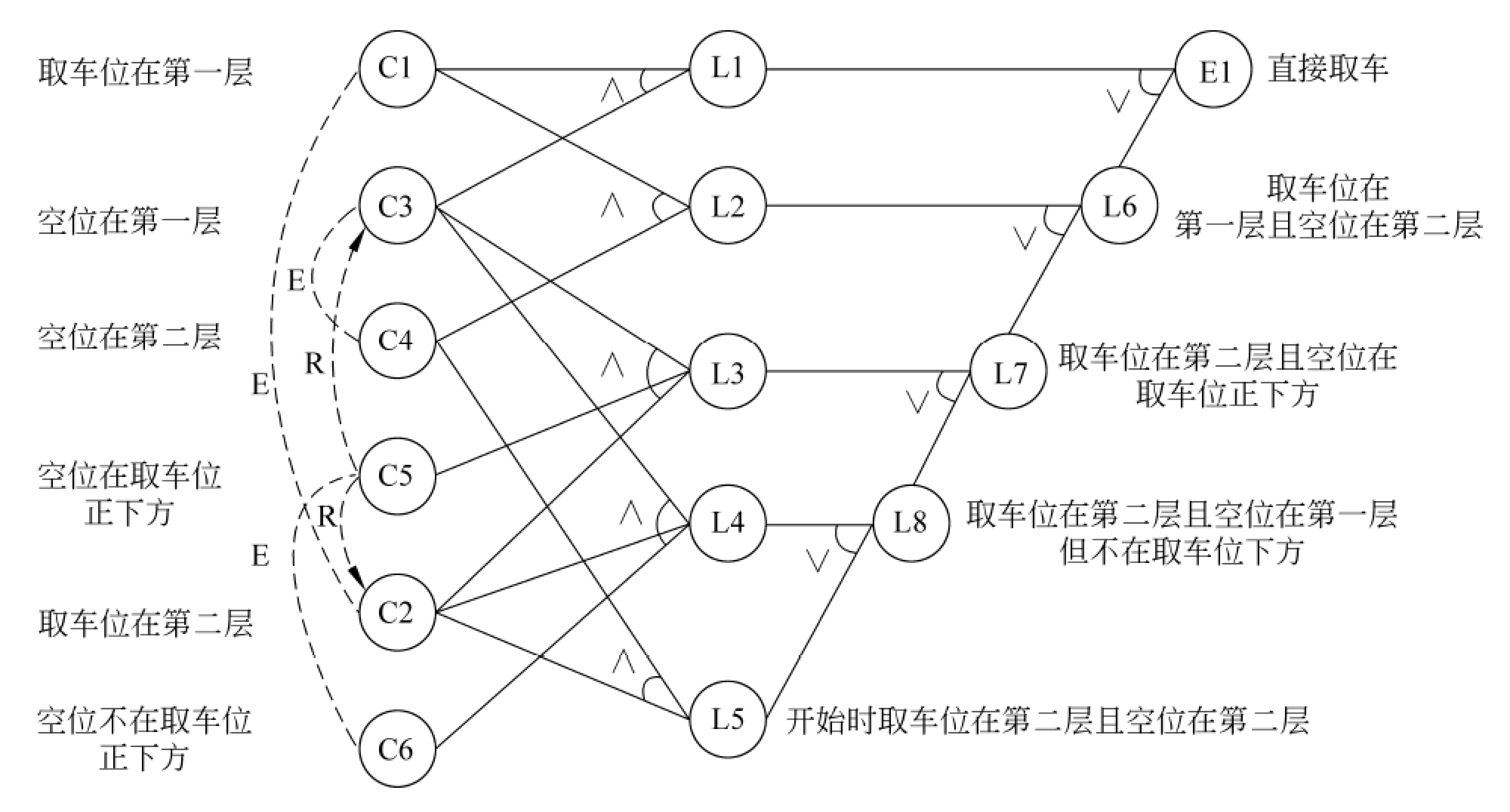


图 2-28 取车过程的因果图

表 2-26 取车问题的决策表

序号			1	2	3	4	5	6	7	8	9	10
输入		C1：取车位在第一层	1	1	1	0	0	0	0	0	0	0
		C2：取车位在第二层	0	0	0	1	1	1	1	0	0	0
		C3：空位在第一层	1	0	0	1	1	0	0	1	0	0
		C4：空位在第二层	0	1	0	0	0	1	0	0	1	0
		C5：空位在取车位正下方	—	—	—	1	0	—	—	—	—	—
		C6：空位不在取车位正下方	—	—	—	0	1	—	—	—	—	—
中间节点	开始状态	L1：开始时取车位在第一层且空位在第一层	1	0	0	0	0	0	0	0	0	0
		L2：开始时取车位在第一层且空位在第二层	0	1	0	0	0	0	0	0	0	0
		L3：开始时取车位在第二层且空位在取车位正下方	0	0	0	1	0	0	0	0	0	0
		L4：开始时取车位在第二层且空位在第一层但不在取车位下方	0	0	0	0	1	0	0	0	0	0
		L5：开始时取车位在第二层且空位在第二层	0	0	0	0	0	1	0	0	0	0
	中间状态	L6：取车位在第一层且空位在第二层	0	1	0	1	1	1	0	0	0	0
		L7：取车位在第二层且空位在取车位正下方	0	0	0	1	1	1	0	0	0	0
		L8：取车位在第二层且空位在第一层但不在取车位下方	0	0	0	0	1	1	0	0	0	0
输出		E1：直接取车	1	1	0	1	1	1	0	0	0	0

通过上面的决策表可以设计相关的测试用例,如表 2-27 所示。

表 2-27 取车问题的测试用例

用例编号	测 试 用 例	预期输出
1	开始时取车位在第一层且空位在第一层	直接取车
2	开始时取车位在第一层且空位在第二层	直接取车
3	开始时取车位在第二层且空位在取车位正下方	直接取车
4	开始时取车位在第二层且空位在第一层但不在取车位下方	直接取车
5	开始时取车位在第二层且空位在第二层	直接取车

第3章 基于控制流的测试

不同需求的实现代码显然是不同的，即使同一个需求也存在完全不同的代码实现方法。不同的代码包含不同的控制流，包括顺序、分支、循环等。白盒测试是和黑盒测试相对应的一种方法，依据不同的控制流覆盖来设计测试用例。覆盖准则包括语句覆盖、判定覆盖、条件覆盖、修正的条件判定覆盖、路径覆盖等。本章重点介绍基于控制流产生测试用例的方法和技巧。

3.1 概述

在实际工作中，经常有人会问，有了黑盒测试以后，为什么还需要进行白盒测试？对于同一个问题需求，存在不同的实现方法。不同的实现方法，可能引入的缺陷是完全不一样的。而黑盒测试无法充分反映出该特征。

下面以斐波那契(Fibonacci)数列为例来描述其实现的不同。公元前13世纪意大利数学家斐波那契的《算盘书》中描述了著名的兔子问题：假定一对大兔子每月能生一对小兔子(一雄一雌)，且每对新生的小兔子经过一个月可以长成一对大兔子，如果不发生死亡，且每次均生下一雌一雄，问由一对刚出生的小兔开始，一年后共有多少对大兔子？

依据该特征，每个月的兔子数量实际上是如下的数列：

0,1,1,2,3,5,8,13,21,34,55,89,144,233,…

该数列就是著名的斐波那契数列，从第三项开始每一项都是前两项之和。写成一般形式为：$f_0=0, f_1=1, \cdots, f_n=f_{n-1}+f_{n-2}$。

斐波那契数列至少存在三种不同的实现方法：①朴素递归；②自底向上动态规划；③数学归纳法。

(1) 朴素递归：自顶向下求解问题，导致了大量的重复求解。以求第5个斐波那契数列为例，其求解过程如图3-1所示，若将第n位斐波那契数列记为Fib(n)；在求解Fib(5)

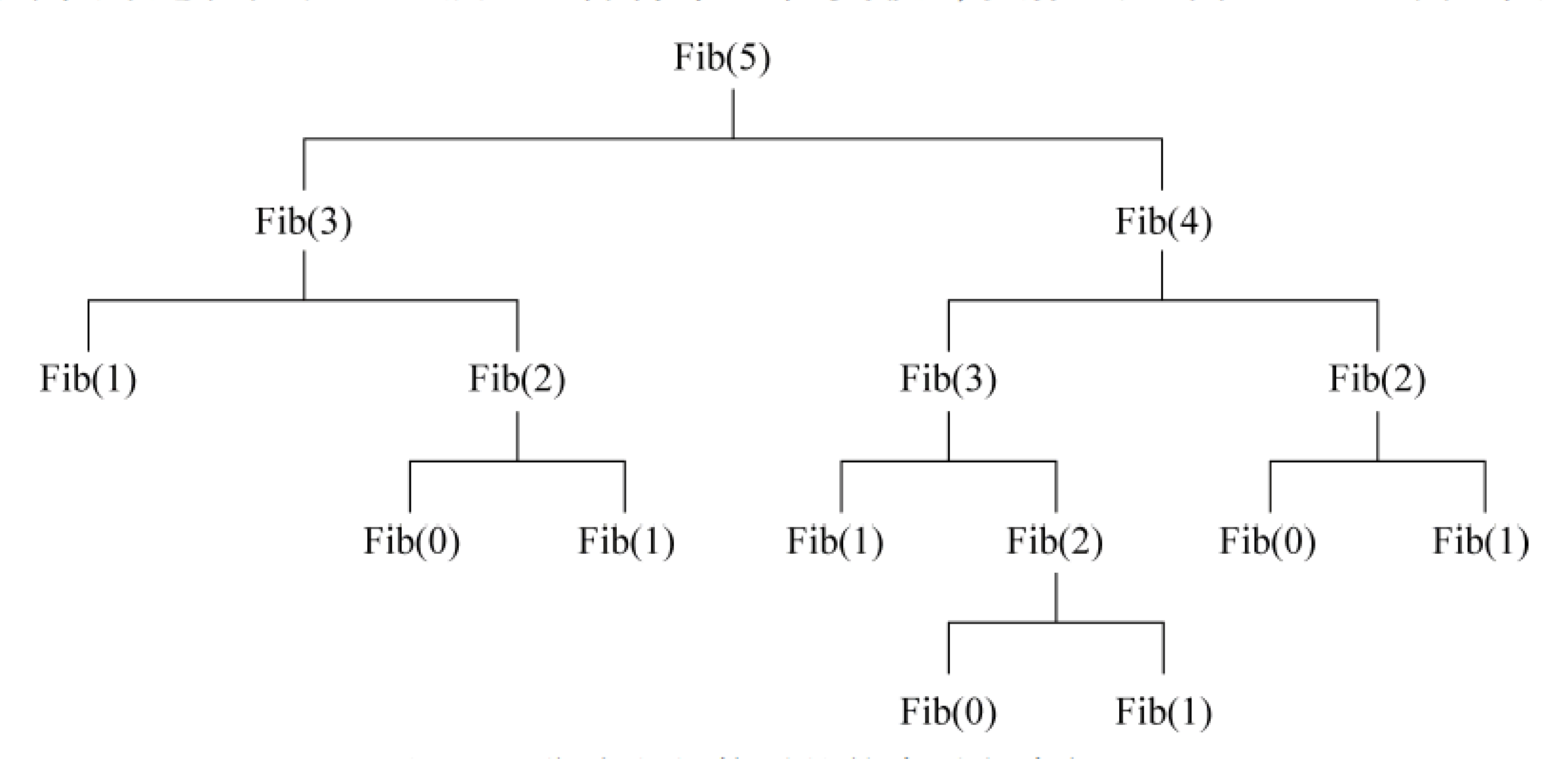


图3-1 斐波那契数列的朴素递归求解过程

时，需要同时计算 Fib(3)和 Fib(4)，而在计算 Fib(4)时，又要重新计算 Fib(3)。

详细代码见程序 3-1，其核心内容为直接递归调用。

程序 3-1　斐波那契的朴素递归的实现程序

```
#程序 1
#fibonacci1.py
def fibonacci1(i):
    if (i<0):
        raise;
    if (i==0):
        return 0;
    if (i==1):
        return 1;
    else:
        return fibonacci1(i-1)+fibonacci1(i-2);

print  fibonacci1(12);
```

(2) 递推法：先求出 Fib(0)和 Fib(1)，然后根据 Fib(2)=Fib(0)+Fib(1)求出 Fib(2)，以此类推，求出所需要的斐波那契数列。该算法的核心内容是一个循环体，采用正向递推的动态规划，在实现过程中利用一个链表保存前面已经求出的结果，所以其所有的中间结果只要计算一次，如图 3-2 所示。

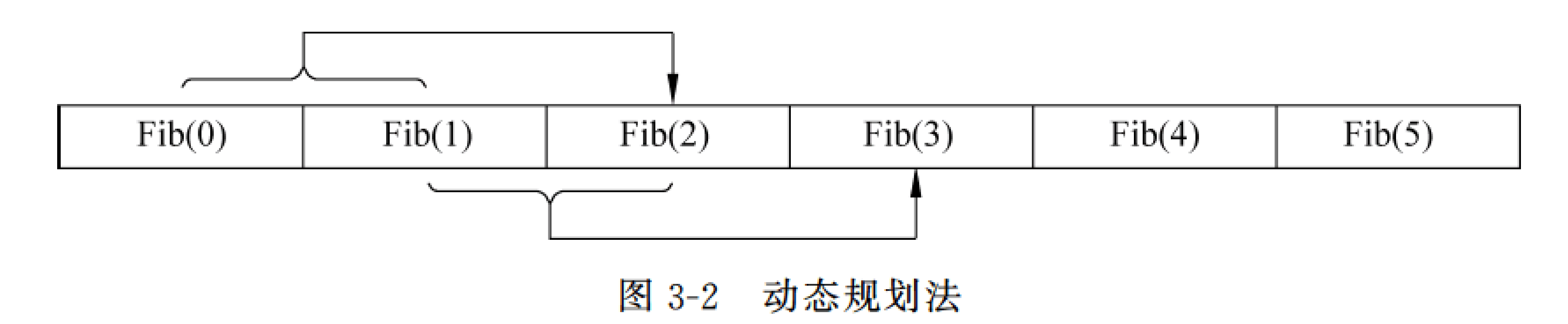


图 3-2　动态规划法

动态规划的详细代码如下。

程序 3-2　斐波那契的动态规划的实现程序

```
#程序 2
#fibonacci2.py
def fibonacci2(i):
    fibArray=[];
    if (i<0):
        raise;
    if(i==0):
        return 0;
    if (i==1):
        return 1;
    else:
        fibArray.append(1);
        fibArray.append(1);
```

```
    for j in range(2,i):                  #Python 下标从 0开始
        fibArray.append(fibArray[j-1]+fibArray[j-2]);
    return fibArray[len(fibArray)-1];

print  fibonacci2(12);
```

(3) 利用数学归纳法求解。其基本原理是,利用 $f_1=0,f_2=1,\cdots,f_n=f_{n-1}+f_{n-2}$ 之间的数学关系,构建其数学的递推关系。

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n+F_{n-1} & F_n \\ F_{n-1}+F_{n-2} & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$$

由此可以利用矩阵关系来表达递推关系。

$$\begin{aligned} \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} &= \begin{bmatrix} F_n+F_{n-1} & F_n \\ F_{n-1}+F_{n-2} & F_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 \\ &= \begin{bmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \\ &= \cdots \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \end{aligned}$$

有了上述数学关系,将求斐波那契的数列转换成为矩阵的幂运算。而对于幂运算又可以采用快速幂方式进行优化。如程序 3-3 所示。其中,power(a,n)用来实现矩阵的幂运算。

程序 3-3　斐波那契的数学归纳法的实现程序

```
#程序 3
#fibonacci3.py
import numpy as np;

a0=[[1,1],[1,0]];

def power1(a,n):
    b=[[1,0],[0,1]];                  #单位矩阵
    while(n>0):                       #快速幂算法求矩阵幂
        if(n & 1):b=np.dot(b,a);      #numpy.dot 矩阵乘法
        a=np.dot(a,a);
        n>>=1;                        #左移
    return b;

def fibonacci3(i):
    if(i<0):
```

```
        raise;
    if(i==0):
        return 0;
    if(i==1):
        return 1;
    else:
        temp=power1(a0,i);
        return temp[1][0];

print  fibonacci3(12);
```

显然对于上面的同一个求解问题，三个程序的基本原理、实现的技术要求都完全不一样。朴素递归的核心内容是通过递归所产生的状态树，而递归程序测试的核心要点是检查递归是否正确，终止条件是否正确。动态规划方法通过一个循环语句，实现正向传递。其测试的核心是循环的终止条件，动态规划的传递方程是否正确。带有快速幂的数学归纳法是所有算法中效率最高的，其测试的重心是求矩阵快速幂相关条件。因为对于不同复杂度的代码逻辑，可以衍生不同执行路径，只有适当的测试方法，才能高效地找出代码中存在的缺陷。

基于控制流的软件测试充分利用被测单元内部的控制信息，允许测试人员针对程序内部逻辑结构及有关信息来设计和选择测试用例，对程序的逻辑路径进行测试。其考虑的是测试用例对程序内部逻辑的覆盖程度，基本目标是覆盖程序中的每一条路径。但是由于程序中一般含有循环，所以路径的数目极大，下面以程序 3-4 为例简要说明。

程序 3-4　带有三路分支的循环程序

```
#loop-condition-test.py
def looptest(a,b,x,n):
    for i in range(0,n,1):            (1)
        if (a[i]>b[i]):               (2)
            x[i]=x[i] * 2;            (3)
        elif (a[i]=b[i]):             (4)
            x[i]=x[i]+2;              (5)
        else:                         (6)
            x[i]=x[i]+5;              (7)
    return x;                         (8)
```

在这段程序的循环语句中，存在两个判断，共形成了三个分支，同时循环本身存在一个循环终止的条件判断，如图 3-3 所示。在这个图中，循环变量的递增并没有独立的语句，为了表达清晰，将其单独画出。

将程序中存在实际含义的语句分别加以编号，单独的 else 由于依附于 if 语句，是 if 语句的一个分支，所以不单独编号。当循环次数为 1 时，由于 a[0]和 b[0]的值不同，存在三种不同路径。考虑循环的直接终止条件，共有 4 种不同的路径，如表 3-1 所示。

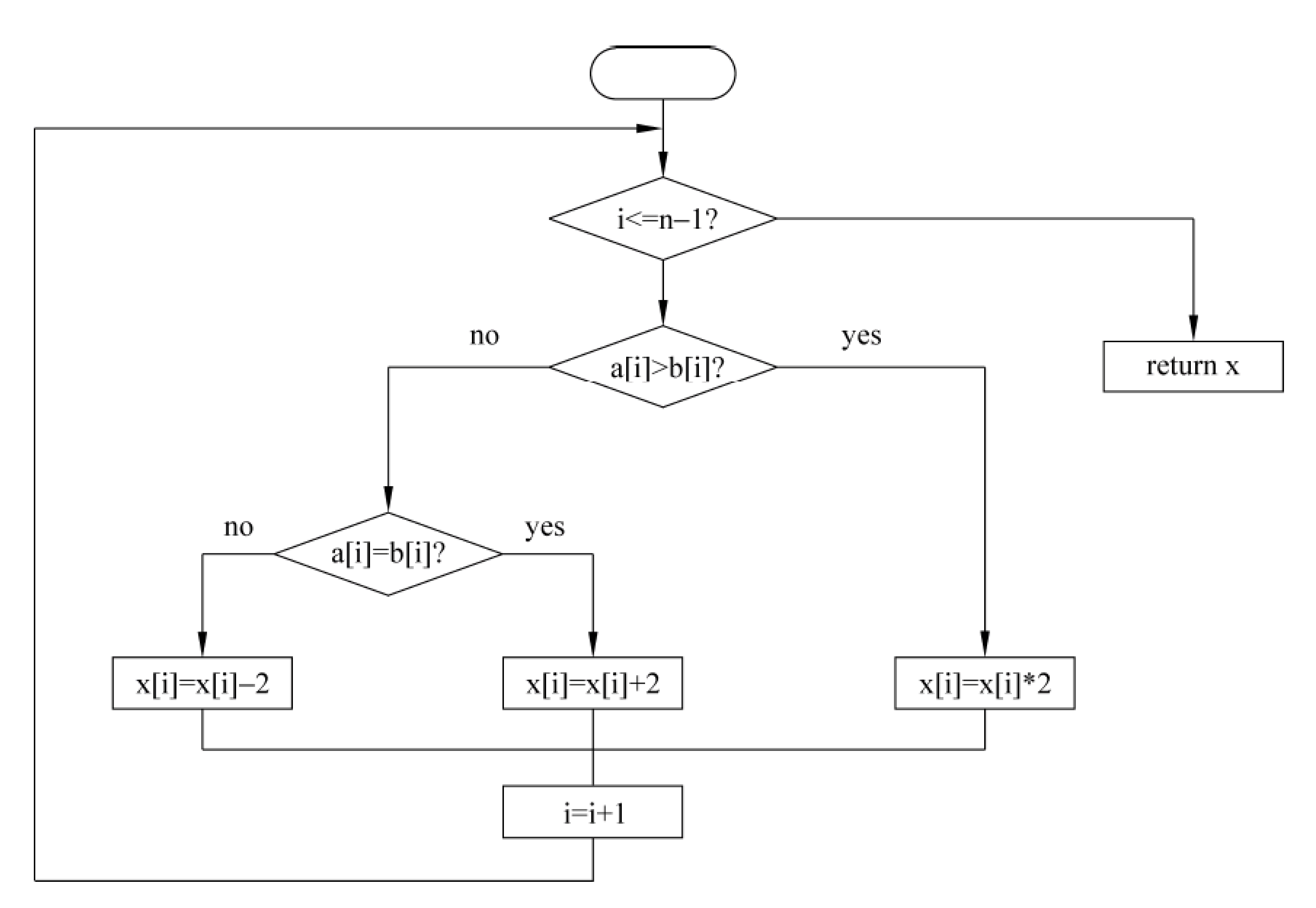


图 3-3 带有三路分支的循环流程图

表 3-1 循环次数为 1 时的路径

编号	条 件	路 径	编号	条 件	路 径
1	a[0]>b[0]	(1)、(2)、(3)、(8)	3	a[0]<b[0]	(1)、(2)、(4)、(7)、(8)
2	a[0]=b[0]	(1)、(2)、(4)、(5)、(8)	4	循环结束	(1)、(8)

可以表达为三种分支和一种循环结束路径所构成：

$$4=3^1+1$$

当循环次数为 2 时，第一次、第二次均可能存在三种不同的路径选择。那么产生的路径分别由第一次判断条件和第二次判断条件所共同决定，如表 3-2 所示。

表 3-2 循环次数为 2 时的路径分析

编号	两次不同的条件	路 径
1	a[0]>b[0],a[1]>b[1]	(1)、(2)、(3)、(1)、(2)、(3)、(8)
2	a[0]>b[0],a[1]=b[1]	(1)、(2)、(3)、(1)、(2)、(4)、(5)、(8)
3	a[0]>b[0],a[1]<b[1]	(1)、(2)、(3)、(1)、(2)、(4)、(7)、(8)
4	a[0]=b[0],a[1]>b[1]	(1)、(2)、(4)、(5)、(1)、(2)、(3)、(8)
5	a[0]=b[0],a[1]=b[1]	(1)、(2)、(4)、(5)、(1)、(2)、(4)、(5)、(8)
6	a[0]=b[0],a[1]<b[1]	(1)、(2)、(4)、(5)、(1)、(2)、(4)、(7)、(8)
7	a[0]>b[0],a[1]>b[1]	(1)、(2)、(4)、(7)、(1)、(2)、(3)、(8)
8	a[0]>b[0],a[1]=b[1]	(1)、(2)、(4)、(7)、(1)、(2)、(4)、(5)、(8)
9	a[0]>b[0],a[1]<b[1]	(1)、(2)、(4)、(7)、(1)、(2)、(4)、(7)、(8)
10	循环结束	(1)、(7)

其路径分别为9条分支路径和1个循环结束分支所构成：

$$10 = 3^2 + 1$$

由以上的分析可以知道，如果由 n 次循环所构成的路径总数为 L，则

$$L = 3^n + 1$$

假设 $n=30$，那么其所产生的路径总数为：

$$L = 3^{30} + 1 = 205\ 891\ 132\ 094\ 650$$

假设机器每秒钟能够执行1000条路径测试，1年转换成为以秒为单位：

$$1\text{ 年} = 365 \times 24 \times 3600\text{ 秒} = 31\ 536\ 000\text{ 秒}$$

那么机器完成所有路径的测试，需要的时间(以年为单位)为：

$$205\ 891\ 132\ 094\ 650 \div 1000 \div 31\ 536\ 000 = 6529(\text{年})$$

而在工业上实际应用软件的规模、复杂度都比程序3-4要复杂得多，要开展全路径覆盖的时间也要远远大于这个时间。由此可知，要对于应用程序的源码进行穷举测试是不可能的。另外，对于源代码的控制流测试也无法发现其他的一些问题：①全路径测试无法发现程序和需求规格说明书的不一致，例如没有实现的需求；②全路径测试无法发现程序因为遗漏路径而出现的错误；③全路径测试无法发现与数据相关的错误。

3.2 图论基础

在代码的控制流结构中，存在判断、多重分支、循环等多重控制流基本结构。在实际的应用中，不仅包含简单控制流结构，而往往是它们之间的复合结构。例如，判断语句中包含判断语句，或者判断语句中包含循环语句，循环语句中嵌套循环语句等，构成极其复杂的控制流结构，这种控制流结构必须采用图才能表达控制流逻辑结构。

图论起源于一些数学游戏难题，如迷宫问题，棋盘上马的行走路线，哥尼斯堡七桥问题等。图论的应用非常广泛，包括运筹学、网络理论、信息论、控制论、博弈论及计算机科学等。1736年，瑞士数学家欧拉发表了图论的第一篇著名论文“哥尼斯堡7桥问题”，如图3-4所示。哥尼斯堡城有一条横贯全城的普雷格尔河，城的各部分用7座桥连接，每逢节假日，有些城市居民进行环城周游，于是便产生了能否“从某地出发，通过每桥恰好一次，在走遍了7桥后又返回到原处”的问题。

哥尼斯堡城的4块陆地部分以 A,B,C 和 D 标记。欧拉把陆地用节点表示，分别标记为 A、B、C 和 D，同时用连接节点的边来表示对应的桥，如图3-5所示，这些节点和边构成了一个图。

图由节点和连接两个节点间的连线组成。一个图可以用三元组 $<V(G), E(G), \Psi(G)>$ 表示，其中：

(1) $V(G)$：非空节点的集合。

(2) $E(G)$：边的集合。

(3) $\Psi(G)$：从边集合 E 到节点无序偶(有序偶)上的函数。

如图3-5所示的哥尼斯堡城图，由于只要求每一座桥均被走一遍，但是对于方向并没有定义。从 A 走向 B，与 B 走向 A 的效果是完全相同的。可以表示为：

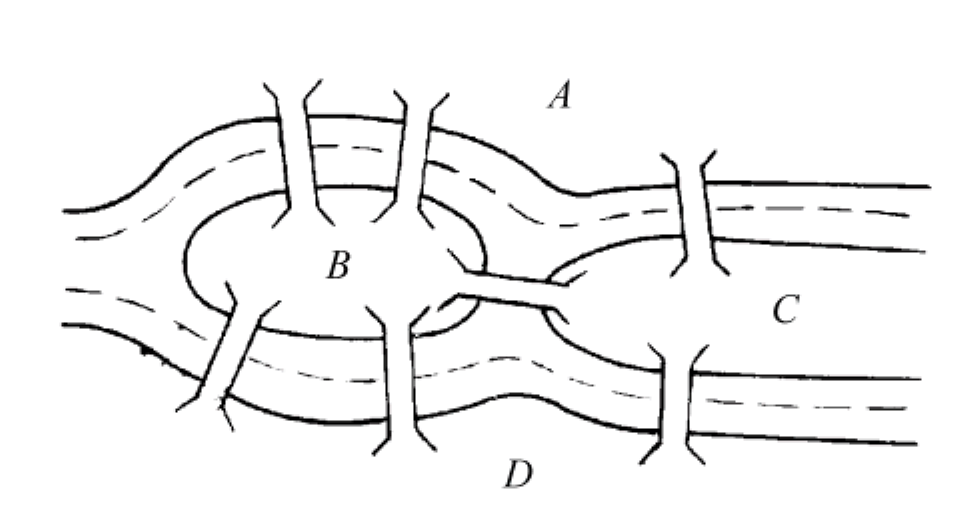


图 3-4 哥尼斯堡 7 桥问题

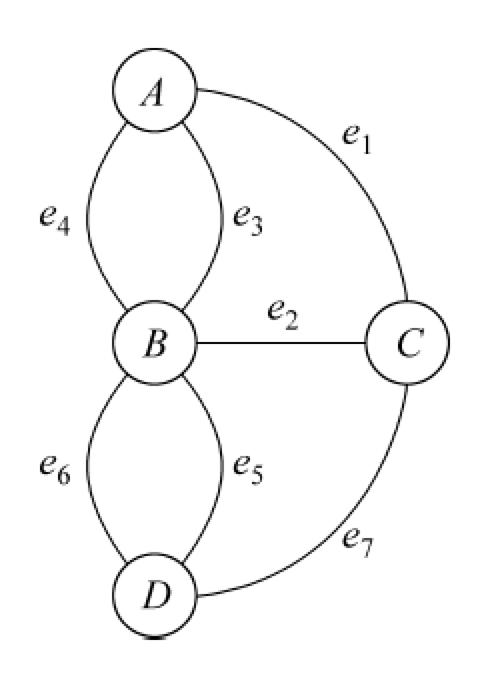


图 3-5 哥尼斯堡 7 桥问题的对应的图

（1）$V(G)=\{A,B,C,D\}$

（2）$E(G)=\{e_1,e_2,e_3,e_4,e_5,e_6,e_7\}$

（3）$\Psi(G)=\{\ \psi(e_1)=(A,C),\psi(e_2)=(B,C),\psi(e_3)=(A,B),\psi(e_4)=(B,A),\psi(e_5)=(B,D),\psi(e_6)=(B,D),\psi(e_7)=(C,D)\}$

关于图的相关定义如下。

无向边：若边 e_l 与节点无序偶(v_i,v_j)相关联。

有向边：若边 e_l 与节点有序偶$<v_i,v_j>$相关联。其中，v_i 为 e_l 的起始节点，v_j 为 e_l 的终止节点。

无向图：若图中所有的边都是无向边，则该图称为无向图。

有向图：若图中所有的边都是有向边，则该图称为有向图。

图 3-6 是一个无向图的示例。在该图中，$V(G_1)=\{A,B,C,D,E\}$，$E(G_1)=\{e_1,e_2,e_3,e_4,e_5,e_6,e_7\}$，所有的边都是无向边，和一条边相关联的是无序偶对。例如，e_5 关联了顶点 D 和 E，对于 e_5 来说，从 D 到 E 和从 E 到 D 效果是完全相同的。

图 3-7 是一个有向图示例，$V(G_2)=\{A,B,C,D,E\}$，$E(G_2)=\{e_1,e_2,e_3,e_4,e_5,e_6,e_7\}$，在这个图中所有的边都是有向边，这些边具有方向性。例如，e_3 是关联的 A 和 D，A 为 e_3 的起点，D 是 e_3 的终点，A 到 D 和 D 到 A 是不同的。例如，用边表示交通时间，从山底向山顶所需要的时间和从山顶向山底所需要的时间不同，必须区分其方向，讨论才具有实际意义。

邻接点：与一条有向边/无向边关联的两个节点。

节点的度：与节点 v $(v\in V)$关联的边数，称作该节点的度数，记为 $D(v)$。

有向图的出度：在有向图中的节点 v，以 v 为始点的边的条数称为节点 v 的出度，记为 $D_{out}(v)$。

有向图的入度：在有向图中的节点 v，以 v 为终点的边的条数称为节点 v 的入度，记为 $D_{in}(v)$。

有向图的度：有向图中某一个节点的度为节点的出度与入度之和，标记为 $D(v)=D_{out}(v)+D_{in}(v)$。

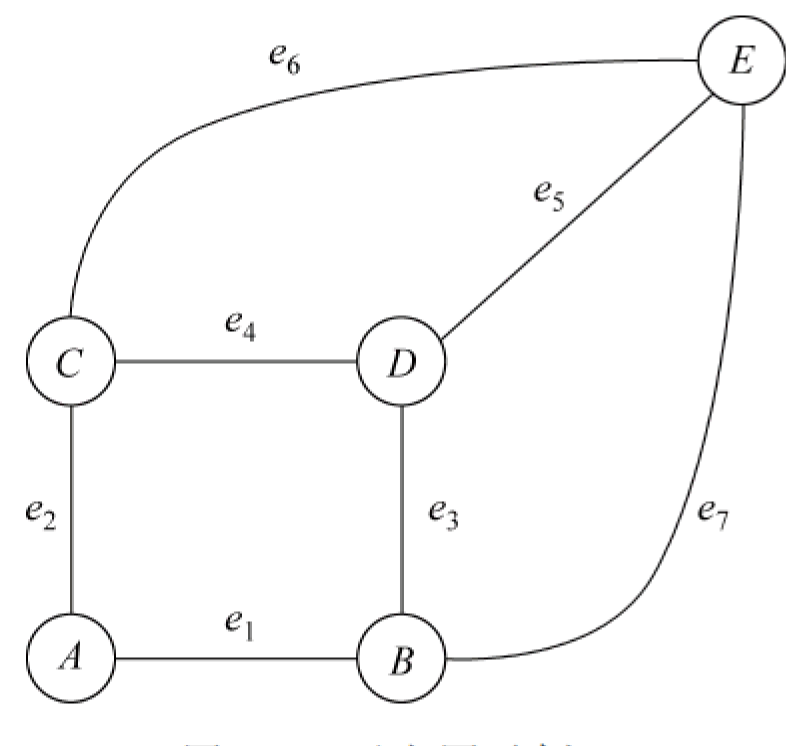


图 3-6 无向图示例 G_1

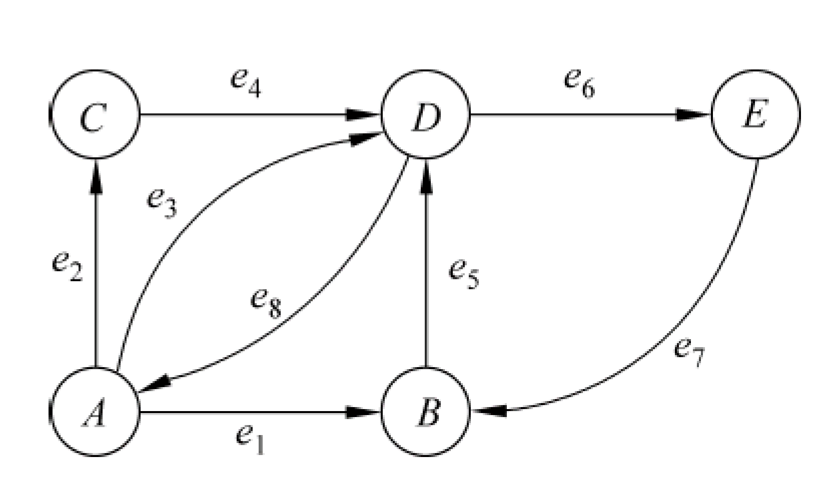


图 3-7 有向图示例 G_2

图的通路：给定图 $G = \langle V, E, \psi \rangle$，设 $v_0, v_1, \cdots, v_n \in V$，$e_1, e_2, \cdots, e_n \in E$，其中，$e_i$ 是关联于节点 v_{i-1}, v_i 的边，交替序列 $v_0 e_1 v_1 e_2 v_2 \cdots e_n v_n$ 称为连接 v_0 到 v_n 的通路。其中，v_0 为起点，而 v_n 是通路的终点。

图的回路：若一条通路的起点和终点相同，则称该通路为回路。

在图 3-6 中，边 e_6 关联节点 C 和节点 E，节点 C 和节点 E 称为边 e_6 的邻接节点。与节点 A 关联的边分别为 e_1 和 e_2，其度为 2。而节点 B、C、D、E 都有三条边和其关联，所以这几个节点的度都是 3。序列 $Ae_2Ce_4De_6E$ 为节点 A 到 E 的一条通路，同理 $Ae_1Be_4De_6E$ 也是一条节点 A 到 E 的通路。序列 $Ae_2Ce_3De_5Be_1A$ 所构成的回路中，其起点和终点都是节点 A，所以该序列是一条回路。在无向图中，通路和回路都没有方向的概念。回路 $Ae_2Ce_3De_5Be_1A$ 和回路 $Ae_1Be_5De_3Ce_2A$ 是完全相同的。

在图 3-7 中，e_4 的邻接节点为 C 和 D，e_4 和 e_8 的邻接节点都是 A 和 D，但是 e_4 和 e_8 的方向是不一样的。节点 C 只和边 e_4 和 e_2 相关联，并且分别作为边 e_4 的起点和边 e_2 的起点，节点 C 的出度 $D_{\text{out}}(C)=1$，入度 $D_{\text{in}}(C)=1$，度 $D(C)= D_{\text{out}}(C)+ D_{\text{in}}(C)=2$。序列 Ae_3De_6E 构成节点 A 到节点 E 的一条通路。序列 $Be_5De_6E\ e_7B$ 构成了一条通路，其起点和终点都是节点 B。

3.3 流程图结构以及表示

几乎所有的软件编程语言，都提供了具有自身特色的控制流语句。控制流图（可简称流图）是对程序控制流进行简化后得到的，可以更加清晰地表示程序控制流结构。

流程图就是 3.2 节中描述的图。控制流图中包括两种图形符号：节点和控制流线。

（1）节点可代表一个或多个语句、一个处理序列和一个条件判定。

（2）控制流线由带箭头的弧或线表示，可称为边，它代表程序中的控制流。

需要分析每一个语句的内部细节，那么在用流程图表示时，区分起始终止节点、循序执行节点、判断节点、输入输出节点，如图 3-8 所示。

其中，典型的流程图包括顺序、分支、循环三种类型。



图 3-8 包含细节的控制流节点

(1) 顺序结构：对于程序(函数)的所有输入，依次执行程序中的每一条语句，这种结构称为顺序结构。

(2) 分支结构：包括两路分支结构和多路分支结构。两路分支一般采用 IF 语句来实现，可以表示为图 3-9。若需要分析其中的语句细节，可以在判断节点(condition1 节点)上表示判定内部的详细信息。一般而言，condition1 的计算结果为逻辑值，存在真和假两种结果。依据不同的计算结果，分别执行 statement2 或者 statement3 语句或者语句组合。图 3-9 中，节点 *A* 表示判断语句，节点 *B* 和节点 *C* 可能是空语句、简单语句或是由其他简单语句构成的复合语句，节点 *D* 是语句 *B* 和语句 *C* 执行以后的汇聚点。节点 *A* 由 if 判断语句所构成，但是判断结果为真时，执行一条路径，如果判断结果为假时，执行另外一条路径。

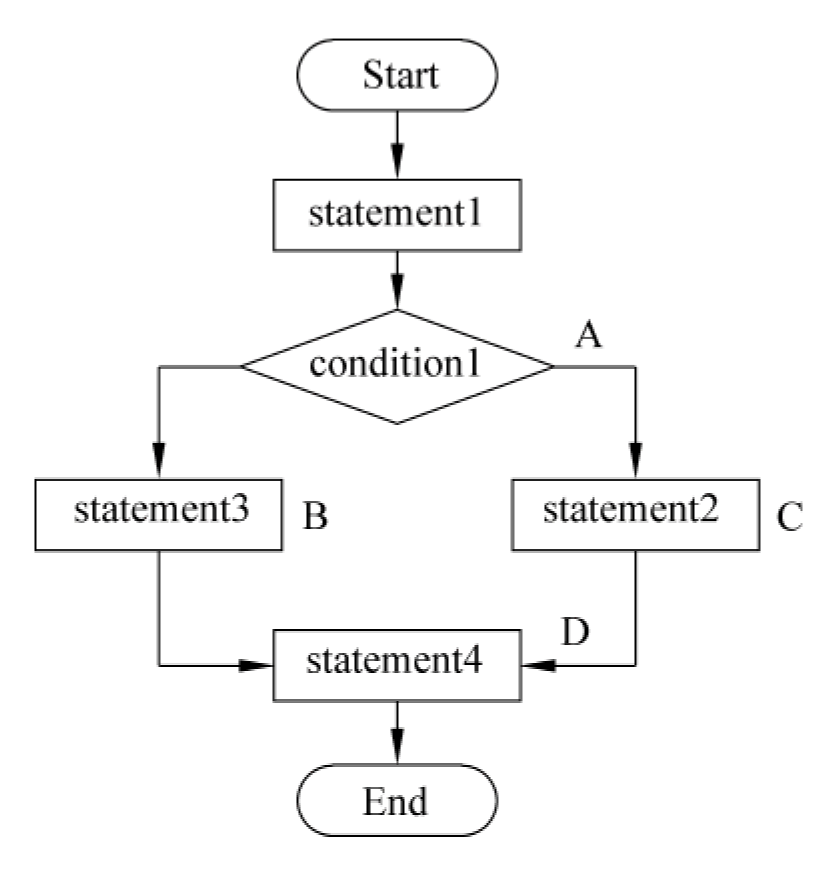


图 3-9 具有两路分支结构

多个 if 组合形成了多路分支。图 3-10 表示由多个 if 语句构成的多路分支结构。

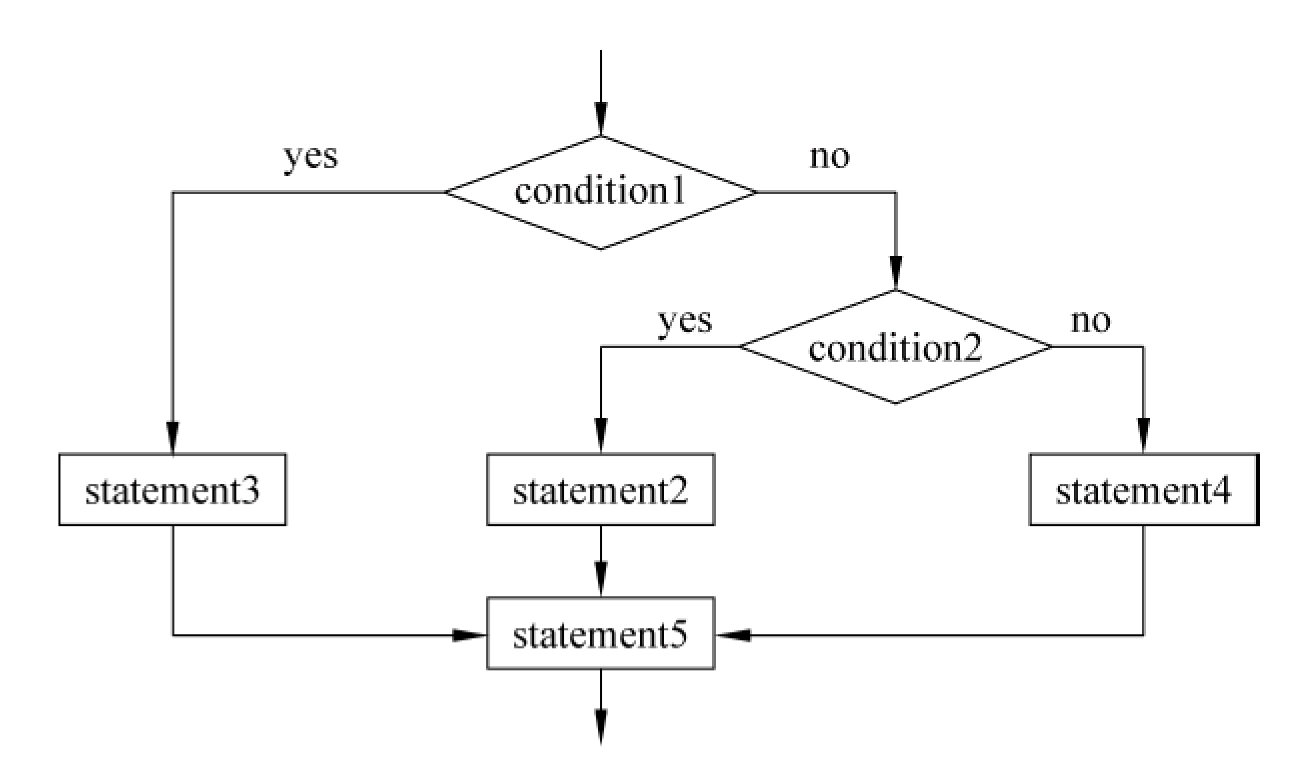


图 3-10 单判断多路分支的结构

多路分支结构在大部分编程语言中采用 case 语句来表示，在有些语言中，图 3-11 表示类似 case 语句所构成的分支结构。多路分支由两个判断组合而构成，其效果与 condition1 和 condition2 两个条件组合而形成复合分支是一样的。这两种结构只是表示上的差别，在流程功能上并没有区别。

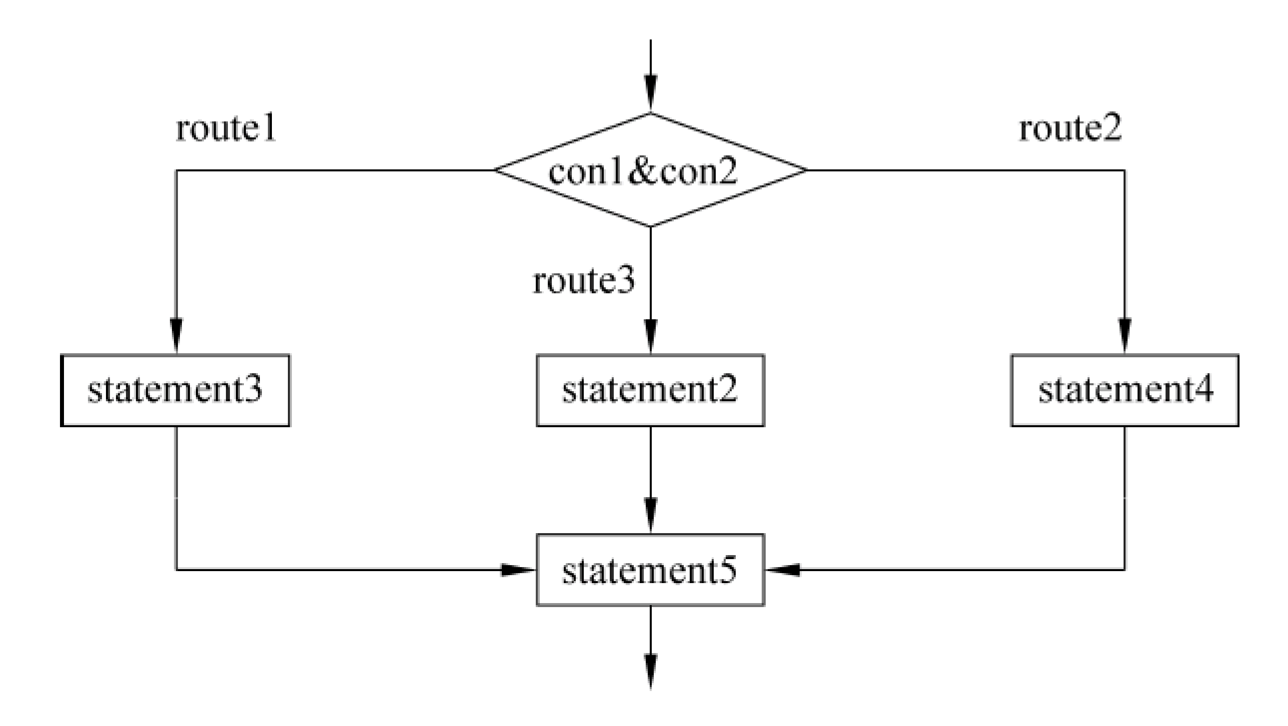


图 3-11　组合判断多路分支的结构

(3) 循环结构：如果一个系列处理要重复执行，往往采用循环结构。例如，对链表的遍历、数据的累加、执行次数的控制等。依据循环终止条件判断所在的位置，可以分成 while 结构和 until 结构。先判断后执行的循环体称为 while 结构，在一般编程语言中都存在的 for 循环语句也是先判断后执行，它和 while 的差异是循环变量的所在位置，在这里统一归纳为 while 结构，如图 3-12(a)所示。先执行一次循环体，然后判断执行条件的循环体称为 until 结构，如图 3-12(b)所示。

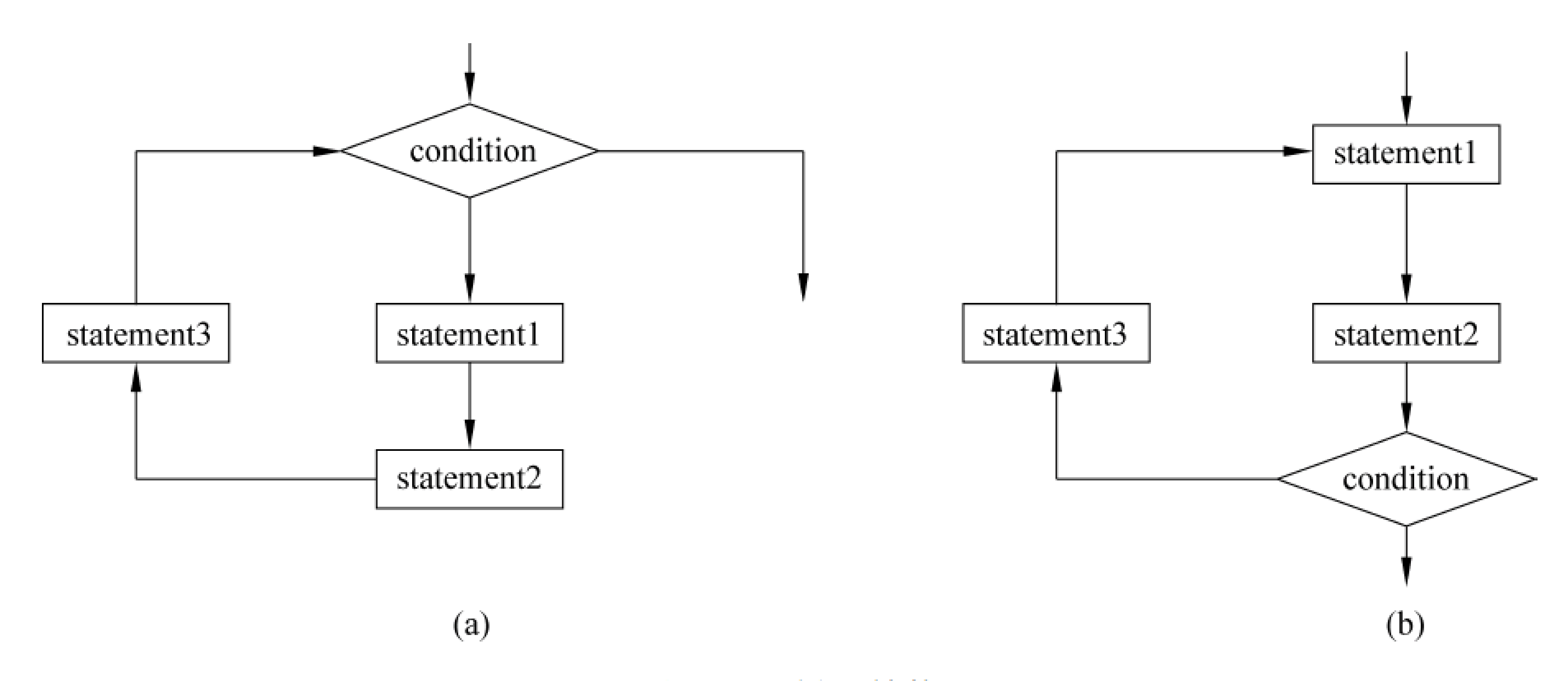


图 3-12　循环结构

3.4　Python 中的条件和判定

在介绍具体的控制流覆盖之前，先介绍条件(Condition)和判定(Decision)的概念。

3.4.1　条件与布尔值认定

条件：不包含布尔操作符的布尔表达式。布尔表达式的运算结果为 True 或者 False。

Python 常见的布尔表达式包括大于(>)、小于(<)、等于(==)、不等于(!=)、是(is)、包含(in)等。其中,特别容易混淆的是==和 is,由于其一切都是对象,Python 对象包含 id、type、value 三个要素,其中,==表示两个标量值是否相同,is 表示两个对象是否为同一个对象。当一个对象有多个引用的时候,并且引用有不同的名称,称这个对象有别名。IDLE 是 Python 的一个典型集成开发环境,在 IDLE 中,以>>>开始的表示输入,而其他的表示输出。图 3-13 在 IDLE 中表示了两者之间的差异。两个变量 a 和 b 的值都是 1.0,它们的值是一样的,但是它们是不同的对象,从 a 和 b 具有的不同 id 也可以说明。因此,a==b 的结果为 True 而 a is b 的结果是 False。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>a=1.0;
>>>b=1.0;
>>>a==b;
True
>>>id(a);
41014648
>>>id(b);
30815936
>>>a is b;
False
```

图 3-13　条件中值相同的判断

要判断两个变量是否为同一个对象,应采用 is 来判断两个变量是否都指向同一个对象的引用。当其中的一个变量值改变时,系统才会创建另一个对象,然后将变量作为引用指向新的对象,如图 3-14 所示。

开始时,变量 a 赋值 2,同时变量 b 作为引用指向 a。a 和 b 都具有相同的值和 id,显然 a is b 返回值为 True。当将 a 的值修改为 3 以后,a 和 b 无法共享相同的值,所以系统为其创建一个新的对象,而 b 仍然指向原来的对象。这时 a is b 返回的是 False。

如果有别名的对象是可变类型的,那么对一个别名个别元素的修改就会影响到另一个,但是它们仍然是同一个对象。图 3-15 中,变量 a 指向列表[1,2,3],是一个指向一个列表的引用,同时 b 指向同一个列表[1,2,3]。变量 a 和变量 b 都是一个变量,当修改变量 b 所指向对象的第一个元素时,将其修改为 5,列表变为[1,5,3]。变量 a 和变量 b 还是指向同一个列表的对象,所以 a 的第二个元素值也随之修改。当把变量 a 指向一个新的列表[4,5,6]时,变量 a 和变量 b 不再指向同一个列表。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>a=2;
>>>b=a;
>>>id(a);
31503964
>>>id(b);
31503964
>>>a  is b;
True
>>>a=3;
>>>b
2
>>>id(a);
31503952
>>>id(b);
31503964
>>>a is b;
False
>>>
```

图 3-14　条件中对象相同的判断

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>a=[1,2,3];
>>>b=a;
>>>a
[1, 2, 3]
>>>b
[1, 2, 3]
>>>b[1]=5;
>>>a
[1, 5, 3]
>>>b
[1, 5, 3]
>>>a is b
True
>>>a=[4,5,6];
>>>a
[4, 5, 6]
>>>b
[1, 5, 3]
>>>a is b
False
>>>
```

图 3-15　可变对象类型是否相同的判定

除了上述讨论的情况外，包括函数在内任何返回值为 True 的都可以认为是一个条件。Python 对于布尔值有一些特别的规定，主要包括：

(1) 任何非零数值或者非空对象都是 True。

(2) 数值零、空对象以及特殊对象 None 都是 False。

3.4.2 判定与短路计算

判定：由零个或者多个条件语句通过逻辑运算符组成，如果同一条件在判定中出现多次，则认为是不同的条件。逻辑运算符包括逻辑与(and)、逻辑或(or)、逻辑非(not)三种。在逻辑与(and)和逻辑或(or)构成的判定表达式中，其返回结果不是简单的 True 或者 False，而是对象。不是运算符左边的对象就是右边的对象。

在逻辑与(and)运算中，Python 从左到右计算每一个条件，并且停留在第一个为 False 的对象上。图 3-16 中给出了具体的例子，在第一个表示式中，2 表示 True，而运算符是逻辑与(and)无法直接确认该判定值，而 3 也是表示 True，这时已经可以确定判定的值，直接将该值作为表达式的值。在 0 and 3 判定中，0 表示 False，其和任何值的与运算的结果都是 False，可以直接确定表达式的值，于是 0 作为整个判定的值。[] and 3 也是同样的情况。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>2 and 3;
3
>>>3 and 2;
2
>>>0 and 3;
0
>>> [] and 3;
[]
>>>3 and [];
[]
>>> [] and {};
[]
>>>
```

图 3-16 逻辑与判定的计算样例

在由逻辑或 or 所构成的判定中，Python 从左到右操作对象，然后返回第一个为真的操作对象。在图 3-17 中的 2 or 3 的判定表达式中，2 是一个非 0 值，被认定为 True，直接判定该值作为判定的表达式，在 or 右边的表达式也不再执行。而在[] or 2 判定中，[]为 False 无法直接确定判定的值，所以继续计算在 or 右边的表达式，并将该值作为判定的值。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>2 or 3
2
>>>3 or 2
3
>>> [] or 2
2
>>>2 or []
2
>>> [] or {}
{}
>>>0 or []
[]
>>>
```

图 3-17 逻辑或判定的计算样例

无论是 and 运算还是 or 运算，Python 在第一个能够确定其表达式的条件上停止运算，并将该条件的值作为整个判定的最终值。对于剩余的部分，不再继续进行计算，这个现象称为短路计算。对于在逻辑运算符两端都是简单的变量或者表达式，不存在问题。但在逻辑表达式中可能会调用函数来完成一定的功能，那么该功能有可能由于短路计算被忽略。短路计算是一种较为容易被忽略的缺陷，并且用人工检查较难发现。

例如，代码段：

```
if  func1() and  func2(): …
```

当 func1()的返回值为 True 时，Python 将不再执行函数 func2()。为了保证两个函数都能够正确地执行，必须在判定语句之前调用它们。建议在执行 and 语句之前先调用它们，然后根据其结果来执行判定。代码段可以修改为：

```
temp1= func1();
temp2= func2();
if func1() and func2();
```

在设计测试时，除了通用的语句、判定、条件、路径覆盖以外，必须考虑到和特定语言相关的规则作为设计的重要补充。

3.5 语句覆盖

软件存在错误的一个重要原因是包含错误的语句没有被测试到。显然，一个语句没有被执行到，那么测试必然无法发现其中包含的错误。在各种控制流结构中，一个最基本的要求就是所有的语句都被测试用例所覆盖。

3.5.1 语句覆盖定义及其测试

如果一个测试用例，使得被测试的文件(类或者函数)中所有的语句至少被执行到一次，那么这种测试覆盖准则被称为语句覆盖。

语句的覆盖程度可以采用语句覆盖率来表示。

语句覆盖率，是指测试用例所覆盖的可执行语句和可执行语句总数的比例。

$$Cov_{statement} = \frac{St_{executed}}{St_{total}}$$

其中：

$St_{executed}$：表示被执行到的语句；

St_{total}：所有的可以执行的语句。

语句覆盖就是度量被测代码中可执行语句的被执行程度。“可执行语句”不包括代码注释、空行等，只统计能够执行的代码被执行了多少。Coverage 是一个用于统计 Python 代码覆盖率的工具，支持 HTML 报告生成，最新版本支持对分支覆盖率进行统计。官方站点为 http://nedbatchelder.com/code/coverage/，目前最新版本为 3.7.1。

其在线安装的命令为：

```
easy_install coverage
```

在 Windows 系列下，离线安装的命令为：

```
coverage-3.7.1.win32-py2.7.exe
```

其基本命令为：

```
coverage<command> [options] [args]
```

最常用的命令有以下几个。

(1) 收集覆盖信息：coverage run

(2) 查看覆盖概要信息：coverage report

(3) 生成 HTML 格式的详细报告：coverage html

(4) 组合多个覆盖信息：coverage combine

若原来的 Python 程序运行命令为：

```
myprog first second
```

其中，myprog 为 Python 程序，first 和 second 为程序命令行参数。

则收集覆盖信息命令为：

```
coverage run myprog first second
```

如图 3-18 所示为一个 Coverage 覆盖率统计结果的样例，其基本程序调用另一个文件中的 sumEven 函数。生成该报告的命令为：coverage run statementcoveragedemo2.py。然后生成 HTML 文件：coverage html statement-coveragedemo2.py。

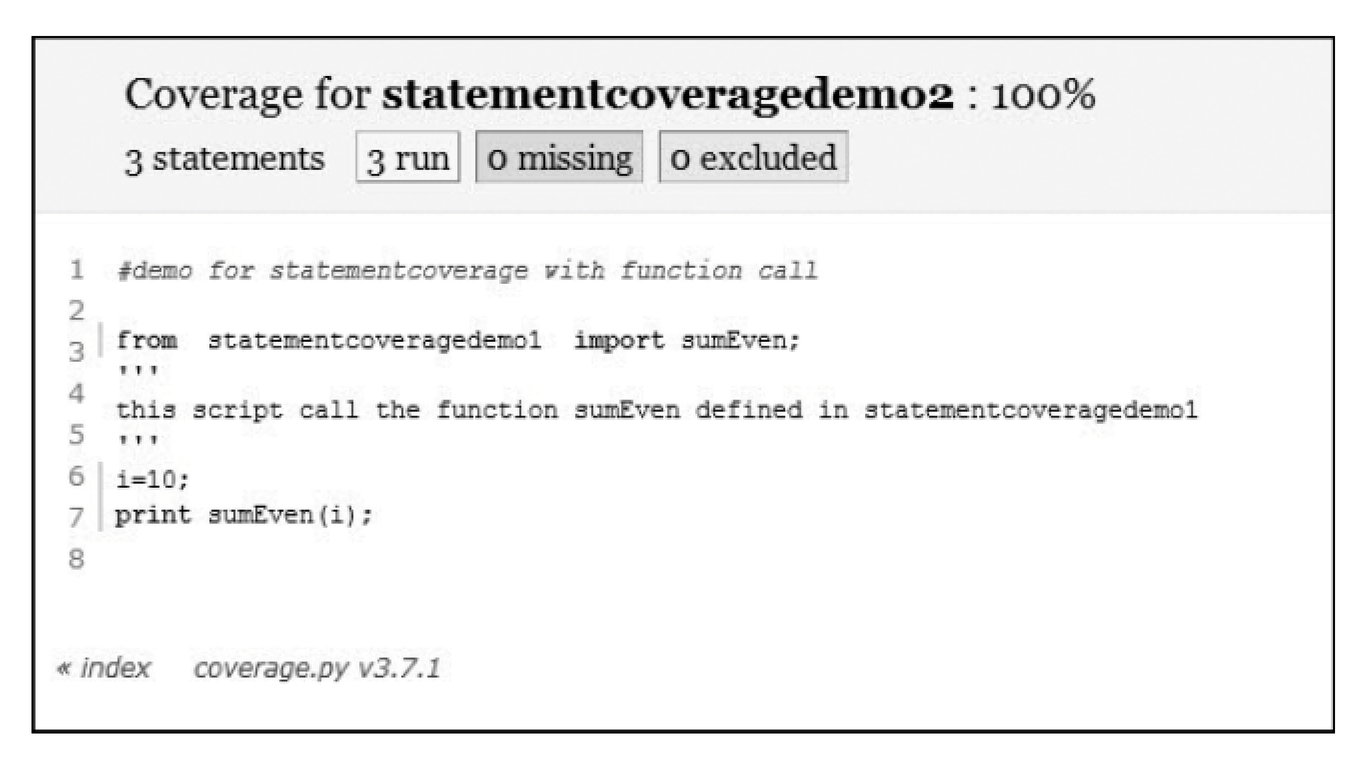


图 3-18　注释和空格不作为可执行语句的统计

Coverage 统计结果为三个可执行语句，从报告可以看出，无论是＃开始的单行注释，以及"的多行注释都没被包含在代码行中间。在这个代码段中，调用另一个文件的函数，Coverage 将其作为一个语句进行统计。

如果需要对 sumEven 函数一起统计，可以利用 run 的-p 参数自动生成多文件的覆盖信息，再利用 combine 命令组合被调用函数的统计结果，生成的统计结果如图 3-19 所示。

Coverage for **statementcoveragedemo1** : 100%

7 statements　7 run　0 missing　0 excluded

```
# demo1
def sumEven(n):
    """
    calculate the sum of even between 0 and n
    """
    sum=0;
    for i in range(n):
        if (i%2==0):
            continue
        else:
            sum+=i;

    return sum;


#if (__name__=="__main__"):
    #print sumEven(10);
```

« index　coverage.py v3.7.1

图 3-19　和 if 对应的 else 并不作为统计结果

在图 3-19 的统计结果中，和 if 对应的 else 语句并没有作为可执行语句，因为其始终是和 if 语句相伴而形成，仅作为另外一个分支，并且是一个确定的分支，并没有产生新的语义和路径。类似的情况，当 else 出现在 while 和 for 循环中时，统计结果类似，如图 3-20 所示。

接下来，以程序 3-5 为例来阐述语句覆盖的基本原理。

Coverage for **untitled-1** : 90%

10 statements | 9 run | 1 missing | 0 excluded

```
def demo2(n):
    sum=0
    for i in range(n):
        if (i%2 ==0):
            sum+=i;
    else:
        if (i % 2==0):
            sum=sum+2;
        else:
            sum=sum+1
    return sum;

print demo2(8);
```

« index   coverage.py v3.7.1

图 3-20　循环语句中的 else 语句

程序 3-5　一个简单的语句覆盖演示函数

```
#state1.py
def  demo1(a,b,x):
    if (a>1 and b==0):
        x=x/a;
    if(x>1 or a==2):
        x=x+1;
    return  x;
```

程序 3-5 对应的控制流图如图 3-21 所示。控制流图中存在两个判定，而每一个判定都存在两个分支，在判定 1 中的分支为(2)和(3)，分支(2)不执行任何语句直接跳转到判定 2 中，分支(3) 执行了 x＝x/a 语句。第二个判定的情况类似。

为了满足语句覆盖的基本要求，只要让测试沿着(1)、(3)、(5)往下执行，就能满足语句覆盖的要求。在本章中，采用 py. test 作为测试用例的基本表达式，关于 py. test 的详细信息可以参见附录 D。在测试程序中，若令输入 a＝2，b＝0，x＝3，函数 demo1 的输出用变量 r 表示，此时 r 应该为 2，其对应的测试程序如程序 3-6 所示。

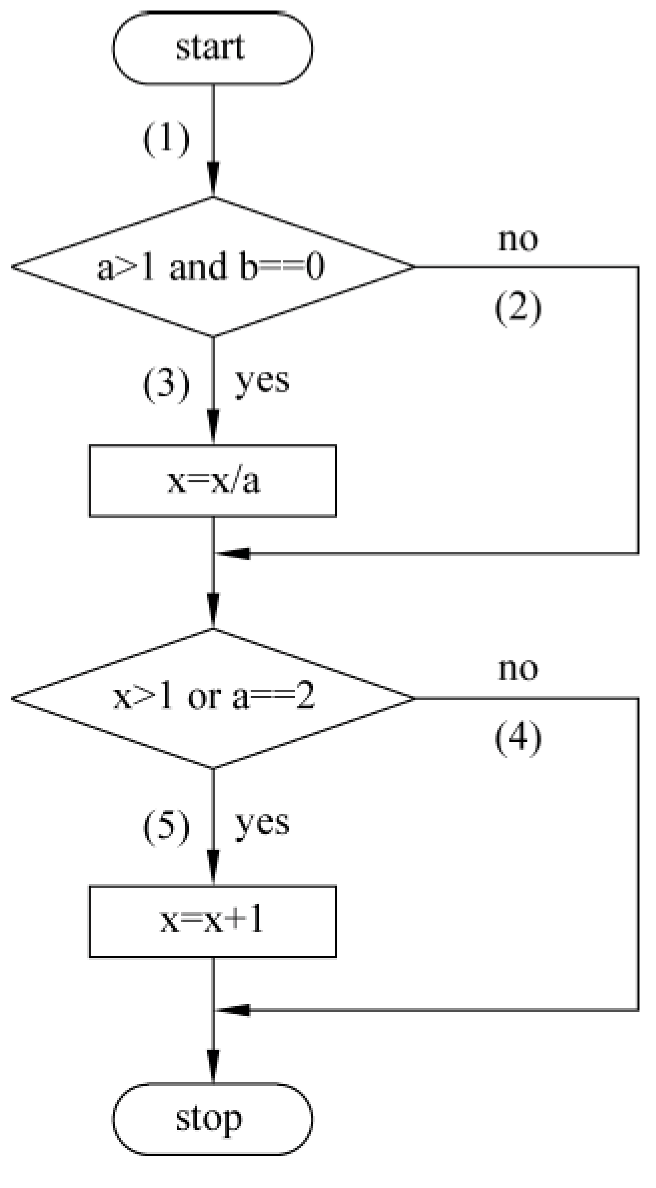


图 3-21　程序 3-5 对应的控制流图

程序 3-6　demo1 对应的测试程序 1

```
#test_state1.py
import  pytest;
```

```
import state1;

@pytest.mark.parametrize("a,b,x,r", [
    (2,0,3,2),
])
def test_demo1(a,b,x,r):
    assert state1.demo1(a,b,x)==r;
```

若输入为 a=3,b=1,x=0,输出 r 为 0,其对应的测试用例如程序 37 所示,那么其执行的路径为(1)、(2)、(4),显然语句 x=x/a 以及语句 x=x+1 都没有被覆盖到。语句覆盖率为:4/6=66.7%

程序 3-7　demo1 对应的测试程序 2

```
#test_state1.py
import  pytest;
import state1;

@pytest.mark.parametrize("a,b,x,r", [
  (3,1,0,0),
])
def test_demo1(a,b,x,r):
    assert state1.demo1(a,b,x)==r;
```

3.5.2　语句覆盖的优缺点

语句覆盖的优点如下。

(1) 检查所有语句。

(2) 结构简单的代码的测试效果较好。

(3) 容易实现自动测试。

(4) 代码覆盖率高。

但语句覆盖无法发现很多逻辑判断相关的问题。例如,在程序 3-5 中,倘若将 if(a>1 and b==0)中的逻辑与 and 误写成逻辑或 or,如程序 3-8 所示。

程序 3-8　demo1 一个包含缺陷的变体

```
#state2.py
def  demo2(a,b,x):
    if (a>1 or b==0):#错误的语句,应该为 if (a>1 and b==0)
        x=x/a;
    if(x>1 or a==2):
        x=x+1;
    return  x;
```

构建如程序 3-9 所示的测试,但是并不能发现该问题。

程序 3-9　程序 3-8 对应的测试用例

```
#test_state2.py
import  pytest;
import state1;

@pytest.mark.parametrize("a,b,x,r", [
  (2,1,0,1),
])
def test_demo2(a,b,x,r):
    assert state2.demo2(a,b,x)==r;
```

同样在循环语句中，也存在尽管测试用例实现了语句覆盖，但是仍无法发现其中隐含的错误。例如，求列表 a 中从第 1 个到第 n 个元素中，其值等于 k 的个数，有个实现如程序 3-10 所示。

程序 3-10　隐含循环次数错误的程序

```
#state3.py

def demo1(a,k):
    num=0;
    n=5;
    for i in range(1,n,1):
        if (a[i]==k):
            num=num+1;
    return num;
```

在 Python 中，range(1,n,1)函数的范围为 1～n－1，不包括 n。正确的循环判断应该为 range(1,n＋1,1)或者 range(0,n,1)。设计一个语句覆盖的测试程序，如程序 3-11 所示，其包含的测试用例实现了语句全覆盖，但是无法发现该错误。

程序 3-11　实现程序 3-10 全语句覆盖的测试程序

```
#test_state3.py
import  pytest;
import state2;

@pytest.mark.parametrize("a,k,r", [
    ([0,3,2,3,4,5],3,2),
])
def test_demo1(a,k,r):
    assert state3.demo1(a,k)==r;
```

由于不同语言的语法规则的差异性，导致对于语句覆盖的理解也存在较大差异。在大部分传统语言中，对于语句体（或者称为符合语句）都有特定的标志。例如，在 C 和 Java 语言中的{}，Pascal 语言中的 BEGIN 和 END 等，而在 Python 中是以缩进来表示语句块，同一缩进级别为同一级别的语句块。开启一个新的缩进需要使用：（冒号），代表下一级别的语句块。在一般情况下，一条语句一行，一行放多个语句，就需要用；来分隔语句。

即使在同一个语言环境中，由于编写的习惯不同，也会导致代码数量的不同。在进行语句覆盖度量前，建议先定义好代码的编写规范。否则，不同规范下得出的语句覆盖率无法进行有效比较。在 Python 语言中，其中最有影响力的编程规范是 PEP（Python Enhancement Proposals），这是一份为 Python 社区提供各种增强功能的技术规格，其网址为 https://www.python.org/dev/peps/。在 PEP 中定义了空格、注释、命名、编程建议等多方面内容，在实践过程中，可以参考 PEP 技术规范。

3.5.3 语句覆盖与死代码

语句覆盖的一个非常重要的应用就是发现代码段中可能存在的死代码。所谓死代码就是无论设计多少个测试用例，执行流程都不能到达的代码。在 Python 中遇到 return 语句以后会直接返回，而不论后面是否存在其他的代码。在程序 3-12 中，print 语句直接放在 return 语句后面，因此 print 语句将永远无法执行到。

程序 3-12 return 语句后的死代码

```
def f(self):
    return 0
    print 'dead code'

f()
```

但是这种缺陷比较直观，一般的静态扫描工具都能够直接发现，甚至在一些 IDE 中都提供了内置死代码缺陷检测功能。图 3-22 给出了 Wing IDE 的关于简单死代码的警告信息："Warning：This code will never be reached"。

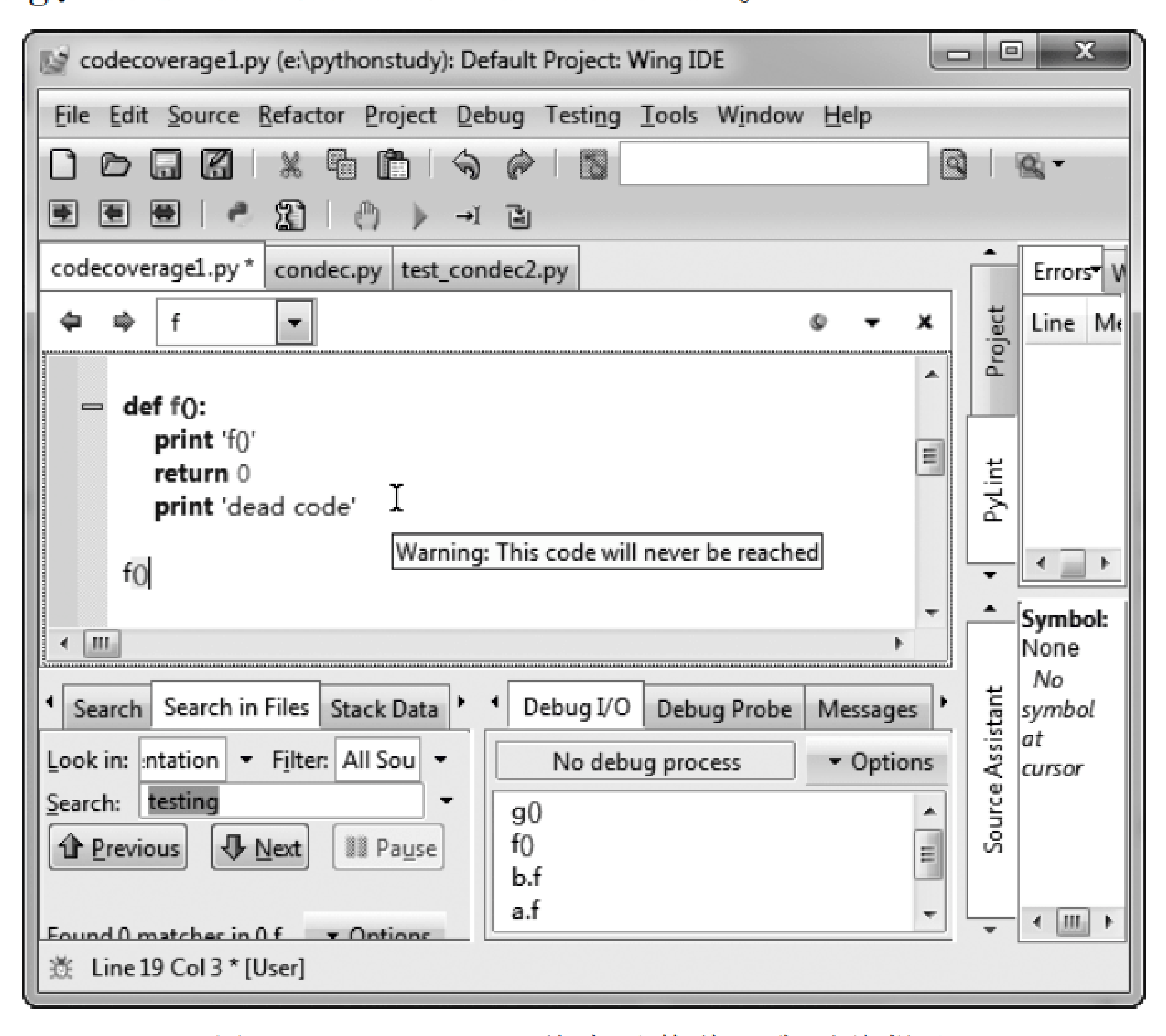


图 3-22 Wing IDE 给出了简单死代码的提醒

程序 3-13 中，return 语句位于 if 语句体的中间，而后面的 print 语句是与 if 语句处于同一个层次。但是由于 not a 在这代码段中是一个永真的表达式，实际上后面的 print 语句是无论如何都不可能执行到的。但是由于其通过一个表达式来判定，仅从静态的语法结构上比较难发现类似的死代码。一般需要通过语句覆盖才能发现类似的缺陷。

程序 3-13　相对比较隐蔽的死代码

```
def g():
    print 'g()'
    a=[];
    if not a:
        return 0
    print 'dead code'
```

另外两个用于控制循环的语句：break 语句和 continue 语句，也比较容易造成死代码。break 语句用于跳出其所在的 for 或者 while 循环，continue 语句用来终止循环语句，即使循环条件没有 false 条件或者序列还没被完全递归完，也会停止执行循环语句。在嵌套循环中，break 语句将停止执行最深层的循环，并开始执行下一行代码。continue 语句用来跳过当前循环的剩余语句，然后继续进行下一轮循环。一般情况下，break 或者 continue 语句都是使用在 if 语句体之中。和 break 或者 continuet 同一缩进层次，并且在它们之后的语句都是永远不可达的。

程序 3-14 中，右边为和需求规格说明书符合的代码，从 1 开始一直到累加到 100，并输出每次累加结果。若累加和大等于 50，也停止累加。print 语句处于 while 模块中和 if 语句并列的层次。由于编程人员出现错误，将 print 语句缩进到 if 语句块中间，并且处于 break 语句的后面，成为死代码。这个程序中，print 语句仅用于使得问题更加清楚而特意设置的，只要直接运行该函数就能发现该问题。在实际应用中可能是计算某一类数值或者执行某一种功能，这个软件缺陷不一定能够如此清楚地发现，但是通过语句执行的覆盖率能够方便找到死代码，发现存在的错误。

程序 3-14　break 语句所构成的死代码

```
#Error function including dead code
def k():
    x=1;
    print 'k()'
    sum=0
    while x<100:
        sum+=x
        if(sum>=50):
            break
            print sum
        x+=1

k()
```

```
#Correctfunction
def k1():
    x=1;
    print 'k1()'
    sum=0;
    while x<100:
        sum+=x
        if(sum>=50):
            break
        print sum
        x+=1

k1()
```

程序3-15右边用于输出1～10之间的奇数，当x为偶数时跳过输出语句直接进行下一次循环，当x为奇数时执行print语句，将x的值输出。而在左边代码中，由于出现错误将print语句放在if语句块中间，并且紧接在continue语句后。在x为奇数时，直接跳过整个if语句体，而当x为偶数时，由于遇到了continue语句，直接跳过print语句执行下一次循环。无论在哪一种情况，都无法执行到print语句。

程序3-15　continue语句所构成的死代码

```
#Error function including dead code
def l():
    x=0;
    print 'l()'
    while x<10:
        x+=1;
        if (x%2==0):
            continue
            print x;
l()
```

```
#Correct function code
def l1():
    x=0;
    print 'l1()'
    while x<10:
        x+=1;
        if (x%2==0):
            continue
        print x;
l1()
```

3.6　判定覆盖

有些测试能达到很高的语句覆盖率，语句覆盖率甚至达到100%，仍无法发现一些缺陷。另外，很多业务逻辑场景和语句的多少没有关系。例如，程序3-16中的函数，包含两个分支，当x不等于0时，其执行99条语句，而当x等于0时，只有1条语句。当设计了一个用例覆盖了x不等于0的所有语句，其覆盖率达到99%。但是从场景上看，遗漏了一半的场景，语句覆盖并不能反映这些场景的特征。

程序3-16　代码不均匀分支框架

```
def demo(x):
    if (x!=0):
        statement1
        ...
        statement99;
    else:
        statement100
```

3.6.1　判定覆盖简介

判定覆盖又称分支覆盖，可以在一定程度上弥补语句覆盖的不足。

判定覆盖就是设计若干个测试用例，使得程序中每个判断的取真分支和取假分支至少经历一次。

判定覆盖率 Dec 定义如下：

$$Dec = \frac{Br_{executed}}{Br_{total}}$$

其中：

$Br_{executed}$：表示已经被测试用例覆盖过的判定输出分支。

Br_{total}：表示所有的判定输出分支总数。

3.6.2 两路分支覆盖

以程序 3-5 为例，在这个例子中，存在两个判定，第一个判定结果会经过路径(2)和路径(3)两种情况，而第二个判定结果会经过路径(4)和路径(5)两种情况。判定覆盖，要求把这些所有的路径全部覆盖。根据这个要求，产生满足判定覆盖的测试用例，如表 3-3 所示。

表 3-3 满足判定覆盖的用例及其覆盖情况

编号	取值情况				判定 1	判定 2	覆盖
	a	b	x	r			
1	2	0	3	2	Yes(True)	Yes(True)	1、3、5
2	3	1	0	0	No(False)	No(False)	1、2、4

显然，这两个测试用例已经覆盖了两个判定的两个结果，这两个测试用例，利用 pytest 框架表示的测试如程序 3-17 所示。

程序 3-17 程序 3-5 的满足分支/判定覆盖的测试代码

```
#test_state1.py
import  pytest;
import state1;

@pytest.mark.parametrize("a,b,x,r", [
    (2,0,3,2),
    (3,1,0,0),
])
def test_demo1(a,b,x,r):
    assert state1.demo1(a,b,x)==r;
```

3.6.3 多路分支覆盖

在实际应用过程中，其分支可能远比前面的多。使用 if-elif-else，或者使用嵌套的 if-else 语句都能实现多路分支。

例如，为了根据学生成绩的不同范围，执行不同的处理，可以采用 if-elif-else 实现。

为了演示的方便，仅返回一个学生成绩的等级符号，如程序 3-18 所示。

程序 3-18　学生等级划分程序

```
#mutipleif.py
def  myRank(score):
    if(score>=90) and (score<=100):
        return "A"
    elif(score>=80 and score<90):
        return "B";
    elif(score>=60 and score<80):
        return "C";
    else:
        return "D";
```

该程序共有 4 路分支，每一个分支处理不同分数段的成绩，若要产生满足分支覆盖准则的测试，必须在 4 个分数段均需遍历。构造 4 个处于不同分数段的分数 55，65，89，90 作为测试的输入。

程序 3-19　学生等级划分程序的测试

```
#test_myRank1.py
import  pytest
from app.mutipleif import myRank

@pytest.mark.parametrize("x,r", [
    (55,"D"),
    (65,"C"),
    (89,"B"),
    (90,"A"),
])
def test_demo1(x,r):
    assert myRank(x)==r;
```

在大多程序设计语言中，除了利用 if 语句实现多路分支以外，还往往提供 switch—case 语句实现多路分支，当变量等于不同的特定值时，选择不同的执行路径。Python 语言不提供 switch—case 语句，可以使用字典实现多路分支功能。

程序 3-20 根据不同的运算符号执行不同的运算功能，这是一个典型的多路分支/判定程序。其核心功能是通过定义字典 calculation 来实现的。

程序 3-20　简易计算器

```
#decision.py
def add(a,b):
    return a+b
def multi(a,b):
    return a* b
def sub(a,b):
```

```
    return a-b
def div(a,b):
    return a/ b#b is non-zero

def calc(type,x,y):
    calculation  = {'+ ':lambda:add(x,y),
                    '* ':lambda:multi(x,y),
                    '- ':lambda:sub(x,y),
                    '/':lambda:div(x,y)}
    return calculation[type]()
```

为了满足这个程序的多路分支测试,必须将+,—,*,/4 种运算符都覆盖到,产生的测试如程序 3-21 所示。

程序 3-21 简易计算器的测试用例

```
#test_myRank1.py
import  pytest
from app.decision import calc

@pytest.mark.parametrize("op,a,b,r", [
    ("+ ",6,3,9),
    ("- ",6,3,3),
    ("* ",6,3,18),
    ("/",6,3,2),
])
def test_demo1(op,a,b,r):
    assert calc(op,a,b)==r;
```

3.6.4 不可达分支

分支覆盖的一个重要应用是发现代码中存在的不可达分支。有些分支,无论设计多少个测试用例,均无法使得该分支的语句体得到执行,那么该分支称为不可达分支。和通过语句覆盖发现死代码类似,判定覆盖可以用于发现一些不可达的分支。

不可达分支,是由于前置路径的相关性导致一些判定的取值始终为 True 或者 False。通常存在下面几种情况。

(1) 值依赖不可达分支:如果一个不可达路径是由判定引用了一个已经赋值的变量而导致该路径不可达,称为值依赖不可达分支。最简单的一种形式是一个变量被赋予一个常数,该常数会导致判定值为一个特定值。

程序 3-22 中具有一个简单的分支,但是其判定结果永远为 False,print 语句处在一个不可到达的分支中。

程序 3-22　值依赖可达分支样例

```
def g():
    print 'g()'
    if[]:
        print 'hello'

g()
```

（2）判定依赖不可达分支：若前面的某一个特定判定导致本判定的取值为一个特定值，称为判定依赖不可达分支。

在程序 3-23 中，当第一个判定值为 True 时，x 的值始终小于 y，第二个判定的值也只有一种情况，其必然为 True。在这种情况下，第二个判定取值无法达到 False，也就是说 b＝2＊a 这条语句，始终是无法执行到的。

程序 3-23　判定依赖不可达分支样例

```
def decdemo(x,y):
    k=5
    if(x<y):
        a=k+x
        if(x<y+1):
            b=a+1
        else:
            b=2*a
    else:
        b=k+y
    return  b;
```

显然，若代码中出现类似的情况，必然存在缺陷，或者说存在很大的缺陷风险。可以通过判定覆盖容易地发现存在的缺陷。

3.6.5　异常处理多分支覆盖

在程序的执行过程中，必然会出现各种各样的异常情况。所谓异常是指改变程序中控制流程的事件。例如，在写文件时可能会遇到磁盘错误、磁盘溢出等情况；在网络通信时，会遇到网络的突然中断。异常发生的时间一般不受软件所控制，异常可能会发生，也可能不会发生。异常发生时，如果没有合适的异常处理机制，将会导致控制流程中断，程序处于完全不受控状态。在各种编程语言中，一般均提供了异常处理机制，确保在异常发生时，能够有恰当的模块针对所发生的异常进行后继处理，使得程序处于受控状态。

常用的方法提供异常处理，用户根据可以预见的异常发生情况，编写合适的异常处理模块，处理发生的异常。在异常发生时，程序中止当前正在执行的代码块，在这个代码块中所有后继代码将不再执行，程序逻辑进入异常处理器，依据用户的异常处理模块执行相应的动作。一个代码段中，可能会同时包含多个异常情况，当发生不同的异常，程序将进入不同的异常处理模块。这种处理结构，和前面的分支结构非常类似。异常处理模块中

的代码，也可能存在软件缺陷，在设计测试时必须覆盖所有的异常处理模块。

在 Python 中，异常会根据错误自动地被触发，也能够由代码触发或者截获。异常是 Python 对象，表示一个错误。当 Python 程序发生异常时需要捕获处理它，否则程序会终止执行。图 3-23 给出了一个异常的例子。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>def add(a,b):
	c=a+b
	print c

>>>add(2,4)
6
>>>
>>>add('abc','efg')
abcefg
>>>
>>>add('a',2)

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in<module>
    add('a',2)
  File "<pyshell#11>", line 2, in add
    c=a+b
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

图 3-23 一个加运算函数遇到的异常

在这个例子中，利用交互环境给出了一个加运算的函数 add 的例子。当调用具有合法加运算的对象时，其输出了加运算的结果。例如，add(2,4)的结果为 6，而 add('abc','efg')的结果为 abcefg。然而当调用 add('a',2)时，在 Python 语言中，'a'和 2 属于字符型和整型，两者之间不能执行运算。系统抛出异常，并给出了调用栈的调用关系，指明该异常的类型为 TypeError，在异常语句后面的 print 语句将不再被执行。Python 系统已经内置了很多标准异常，如表 3-4 所示是部分标准 Python 异常，用户也可以定义特殊的异常。

表 3-4　Python 常见的标准异常

编号	异　　常	含　　义
1	Exception	常规错误的基类
2	ZeroDivisionError	除(或取模)零 (所有数据类型)
3	EOFError	没有内建输入，到达 EOF 标记
4	IOError	输入/输出操作失败
5	NameError	尝试访问一个没有申明的变量
6	KeyError	请求一个不存在的字典关键字
7	ValueError	传给函数的参数类型不正确，比如给 int()函数传入字符串

在 Python 中，捕捉异常使用 try/except 语句。try/except 语句用来检测 try 语句块中的

错误,从而让 except 语句捕获异常信息并处理。一般的异常处理框架如图 3-24 所示。

```
try:
    <语句块 1>              #可能引发异常的代码块
except<expname1>:
    <语句块 2>              #如果在 try部分触发了 expname1 异常
except<expname2>,<数据>:
    <语句块 3>              #如果引发了 expname2 异常,获得附加的数据
except<expname3, expname4>,<数据>:
    <语句块 4>              #如果引发了列表中的异常 (expname3, expname4),
                            #并获得附加的数据
else:
    <语句块 5>              #如果没有异常发生,执行语句 4
finally:
    <语句块 6>              #总会执行的语句块
```

图 3-24 异常处理框架

在 Python 中可以有多个 except 分句,其数量不受限制。else 分句是一个可选项,但是如果出现,则只允许有一次。在 Python 2.5 版本中,finally 可以和 except 和 else 一同出现。若多个异常的处理逻辑是一样的,可以将多个异常放在同一个 except 语句中。

异常处理的一般流程如下:若在语句块 1 的执行时发生了异常,程序跳出语句块 1,执行第一个符合触发异常 except 的语句块。例如在语句块中,触发了 expname2 异常,那么其执行语句块 3。然后执行 finally 中的语句块 6,程序的流程将执行到 try 语句的后继语句。

若在语句块 1 的执行时发生了异常,没有符合的 except,异常将直接执行 finally 中的语句块 6,程序的流程将执行到 try 语句的后继语句。

若语句块 1 的执行未发生异常,如果存在 else 语句,那么程序执行 else 的语句块。然后执行 finally 中的语句块 6,程序的流程将执行到 try 语句的后继语句。

从上述流程上看,异常处理的流程在实际含义上类似于多路分支,图 3-25 给出了一个异常的实际例子。

异常	类似的多路分支含义

defdowork():
    func1()
 	
try:
    dowork()
except expname1:
    expthandle1()
except expname2:
    expthandle2()
	def dowork ():
	    if(func1()==expname1):
	        return expname1;
	    elseif(func1()==exceptname2):
	        return exceptname2;
	
	def mainfunc()
	    if(dowork()==expname1):
	        expthandle1()
	    else if(dowork()==expname2)
	        expthandle2()

图 3-25 异常处理在语义上和 if 分支类似

类似分支覆盖,在异常处理代码中,可以使用异常覆盖准则来设计测试。

异常覆盖准则：设计的测试用例集，使得代码中所有的异常处理块必须被执行过一次。

若将图3-23中的运算放在一个异常处理中，将其代码加运算发在try代码块中，当出现类型错误TypeError异常时，输出'ERROR'的提示信息，如程序3-24所示。

程序3-24 具有异常处理的加运算程序

```
def add(a,b):
    try:
        c=a+b
    except:
        msg= 'ERROR'
    else:
        msg= str(c)
    finally:
        return str(msg)
```

这个程序包含错误时，输出信息为“ERROR”，以便于上层调用函数能够继续处理。为了测试这个程序，构造两个测试用例，其中一个为具有合法加运算的两个变量，另一个是可能出现的非法变量，如程序3-25所示。

程序3-25 具有测试加运算的异常处理的两个用例

```
d#test_exceptadd.py
import pytest
from app.addexcept import add;

@pytest.mark.parametrize("a,b,r", [
    (2,3,'5'),
    ('a',2,'ERROR'),
])
def test_add(a,b,r):
    assert add(a,b)==r;
```

若在一个代码段中，包含多个异常，那么必须覆盖所有的异常。如果需要执行两个数的除法运算，那么可能出现两种异常情况：①传入该函数的不是合法的数值；②除法中的分母为0，如程序3-26所示。在实际应用中，应该是先判断分母是否为零，然后执行运算，在这里仅是为了说明异常处理的演示。

程序3-26 具有异常处理的除法运算

```
def divdemo(a,b):
    try:
        avalue= int(a)
        bvalue= int(b)
        cvalue= avalue/bvalue
    except ValueError:
        result= "NoNumber"
```

```
    except ZeroDivisionError:
        result= "DeviedByZero"
    else:
        result= str(cvalue)
    finally:
        return result
```

显然,对于这个程序,为了满足异常覆盖的要求,需构造三个测试用例。其中一个测试是正确运算的数据,第二个用例为不是合法的数值,第三个用例是分母为零的情况,如程序 3-27 所示。

程序 3-27　除法程序的三个测试用例

```
#test_multipleexcept.py
import  pytest
from app.multipleexcept import divdemo;


@pytest.mark.parametrize("a,b,r", [
    (6,3,'2'),
    ('a',4,'NoNumber'),
    (4,0,'DeviedByZero'),
])
def test_demo1(a,b,r):
    assert divdemo(a,b)==r;
```

3.6.6　复合判定覆盖

工业中的实际应用需求,其复杂性要远远超出上面描述的情况,往往存在多个判定/分支嵌套或者串接的情况。若由多个判定嵌套而形成,称为嵌套型分支;若由多个分支串接起来,称为连锁型分支。更加复杂的是嵌套型分支和连锁型分支的组合。在此仅讨论单纯的嵌套型分支或者连锁型分支情况,其组合情况由读者自行分析。由于两种复合分支覆盖产生的测试用例都比较多,在本节讨论中仅针对流程图给出测试用例的表格表示。

所谓嵌套型分支,是指在一个判定/分支中的一条分支中间,又存在判定/分支的情况。在嵌套型分支中,其最小的测试用例数等于其判定输出分支的最小测试用例数和,若不存在分支情况,那么其用例数记为 1。若一个判定点的测试用例数记为 TC_d,子分支 n 的测试用例数记为 TC_{sdn},那么测试用例数按如下公式计算:

$$\begin{cases} TC_d = 1 \\ TC_d = TC_{sd1} + TC_{sd2} + \cdots + TC_{sdn} \end{cases}$$

显然这是一个递归的过程。图 3-26(a) 是一个嵌套分支的具体例子。在判定 p_1 上,存在三个分支,其中判定 p_2 有三个分支,判定 p_3 有两个分支。那么判定点 p_1 开始计算

的满足分支覆盖的测试用例数,分别由其三个判定输出分支的测试用例数所构成。其总测试用例数为:$TC_{p1}=3+1+2=6$ 条。以 p_1 为分析点,这 6 条路径如表 3-5 所示。

表 3-5 嵌套型分支产生的测试用例

编号	输出分支	路 径
1	p_2	$p_1-p_2-s_2-p_4-p_6$
2		$p_1-p_2-s_3-p_4-p_6$
3		$p_1-p_2-s_1-p_4-p_6$
4	s_7	$p_1-s_7-p_6$
5	p_3	$p_1-p_3-s_5-p_5-p_6$
6		$p_1-p_3-s_6-p_5-p_6$

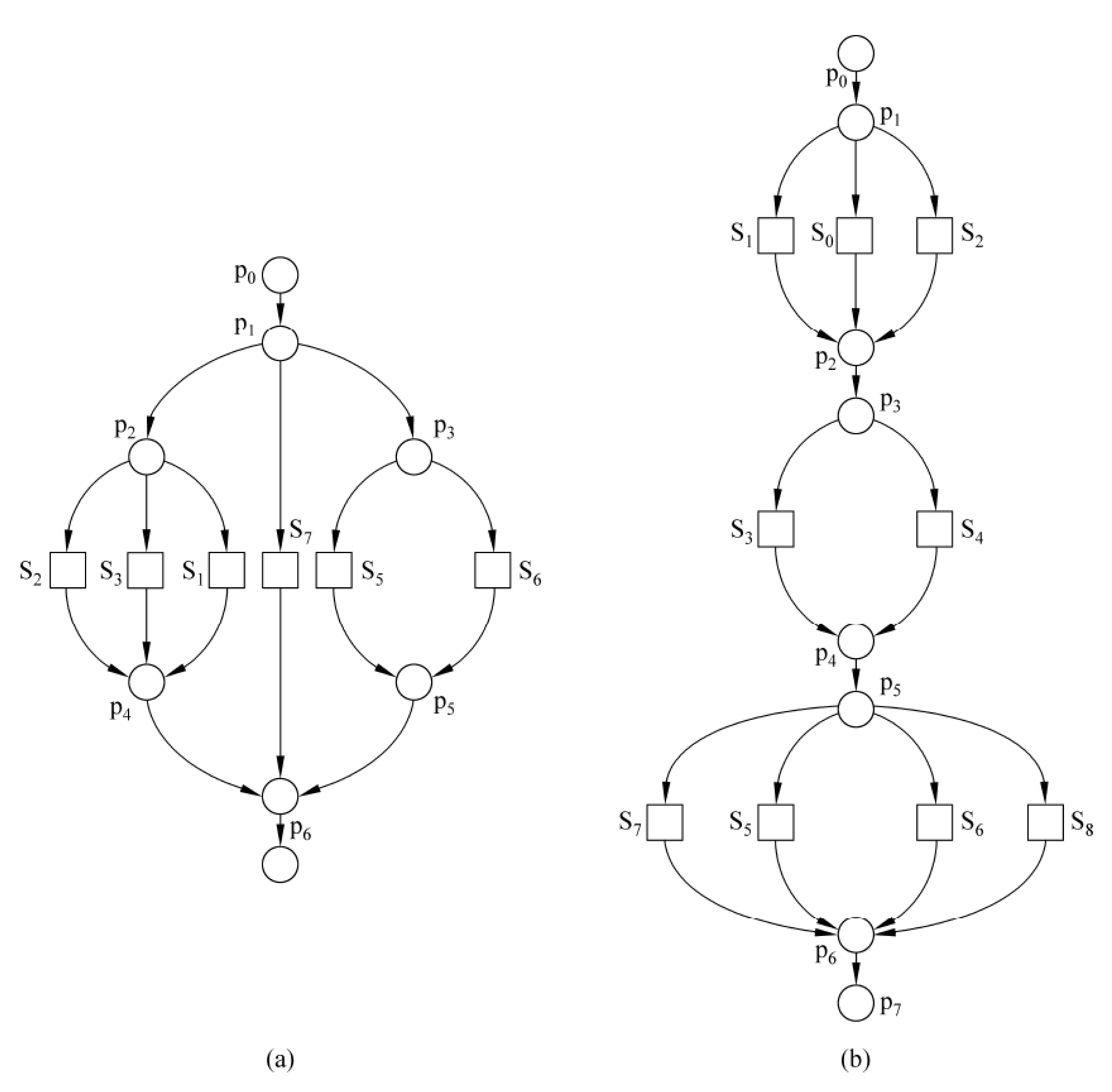


图 3-26 嵌套型和连锁型分支

图 3-26(b)是一个连锁型分支的具体例子,由三个判定串接形成。第一个判定具有三个输出分支,第二个判定具有两个输出分支,而第三个判定具有四个输出分支。在连锁型分支中,其分支覆盖的路径,其最小测试用例数由串接的判定中输出分支中最多的一个

判定所决定。在串接的代码中，测试用例执行的一条路径将经过所有的判定点，每次产生的新路径都经过一个判定点的不同输出，除非该判定点的所有输出分支都已经被覆盖过。在图 3-11(b)中，p_5 判定点的输出分支数最多，所以该代码段的测试用例数共 4 条。从 p_1 开始，从左到右选择每个判定点不同的输出分支，如果所有的分支点都已经被覆盖过，那么从左到右重新选择输出分支。最终产生的测试用例如表 3-6 所示。

表 3-6　连锁型分支所产生的测试用例

编号	路　　径
1	$p_0 - p_1 - s_0 - p_2 - p_3 - s_3 - p_4 - p_5 - s_5 - p_6 - p_7$
2	$p_0 - p_1 - s_1 - p_2 - p_3 - s_4 - p_4 - p_5 - s_6 - p_6 - p_7$
3	$p_0 - p_1 - s_2 - p_2 - p_3 - s_3 - p_4 - p_5 - s_7 - p_6 - p_7$
4	$p_0 - p_1 - s_0 - p_2 - p_3 - s_4 - p_4 - p_5 - s_8 - p_6 - p_7$

在连锁型分支覆盖中，只考虑每一个判定点的输出分支情况。而实际上，对于每一个判定点而言，由不同分支路径进入该判定点，会对其后继产生影响。而前面的讨论过程中，并没有考虑这种影响。引入分支交换覆盖准则：对于每一个判定点，其每一个输入分支和输出分支的组合必须被测试用例覆盖一次。在分支交换覆盖准则中，对于判定点的每一个输入分支，都必须对应所有的输出分支。在图 3-27 中，通过语句 s_0 的输入分支，必须和所有的输出分支$\{s'_0, s'_1, \cdots, s'_{m-1}, s'_m\}$构成输入输出对：

$$\{(s'_0, s'_0), (s'_0, s'_1), \cdots, (s'_n, s'_{m-1}), (s'_n, s'_m)\}$$

对于其他的输入分支，必须和所有的输出分支构成输入和输出关系。若一个判定点具有 n 个输入分支路径，m 个输出分支路径，那么满足分支交换覆盖的测试用例数为 $n \times m$。

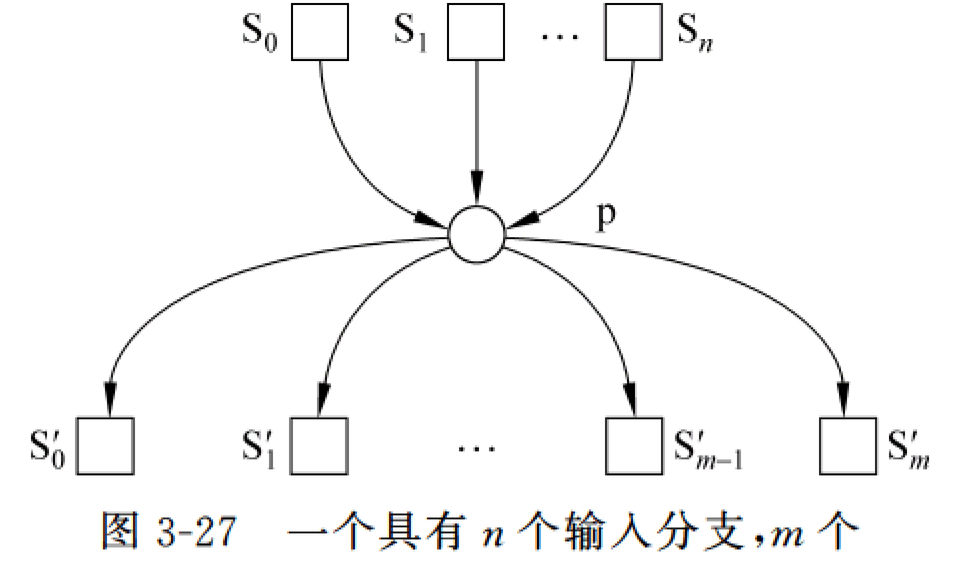


图 3-27　一个具有 n 个输入分支，m 个输出分支的判定点

在图 3-26(b)中，满足分支交换覆盖的测试用例数量为 $3 \times 2 \times 4 = 24$ 条，如表 3-7 所示。

表 3-7　满足分支交换覆盖的测试用例

编号	路　　径
1	$p_0 - p_1 - s_0 - p_2 - p_3 - s_3 - p_4 - p_5 - s_5 - p_6 - p_7$
2	$p_0 - p_1 - s_1 - p_2 - p_3 - s_3 - p_4 - p_5 - s_5 - p_6 - p_7$
3	$p_0 - p_1 - s_2 - p_2 - p_3 - s_3 - p_4 - p_5 - s_5 - p_6 - p_7$
4	$p_0 - p_1 - s_0 - p_2 - p_3 - s_4 - p_4 - p_5 - s_5 - p_6 - p_7$

续表

编号	路　径
5	$p_0-p_1-s_1-p_2-p_3-s_4-p_4-p_5-s_5-p_6-p_7$
6	$p_0-p_1-s_2-p_2-p_3-s_4-p_4-p_5-s_5-p_6-p_7$
7	$p_0-p_1-s_0-p_2-p_3-s_3-p_4-p_5-s_6-p_6-p_7$
8	$p_0-p_1-s_1-p_2-p_3-s_3-p_4-p_5-s_6-p_6-p_7$
9	$p_0-p_1-s_2-p_2-p_3-s_3-p_4-p_5-s_6-p_6-p_7$
10	$p_0-p_1-s_0-p_2-p_3-s_4-p_4-p_5-s_6-p_6-p_7$
11	$p_0-p_1-s_1-p_2-p_3-s_4-p_4-p_5-s_6-p_6-p_7$
12	$p_0-p_1-s_2-p_2-p_3-s_4-p_4-p_5-s_6-p_6-p_7$
13	$p_0-p_1-s_0-p_2-p_3-s_3-p_4-p_5-s_7-p_6-p_7$
14	$p_0-p_1-s_1-p_2-p_3-s_3-p_4-p_5-s_7-p_6-p_7$
15	$p_0-p_1-s_2-p_2-p_3-s_3-p_4-p_5-s_7-p_6-p_7$
16	$p_0-p_1-s_0-p_2-p_3-s_4-p_4-p_5-s_7-p_6-p_7$
17	$p_0-p_1-s_1-p_2-p_3-s_4-p_4-p_5-s_7-p_6-p_7$
18	$p_0-p_1-s_2-p_2-p_3-s_4-p_4-p_5-s_7-p_6-p_7$
19	$p_0-p_1-s_0-p_2-p_3-s_3-p_4-p_5-s_8-p_6-p_7$
20	$p_0-p_1-s_1-p_2-p_3-s_3-p_4-p_5-s_8-p_6-p_7$
21	$p_0-p_1-s_2-p_2-p_3-s_3-p_4-p_5-s_8-p_6-p_7$
22	$p_0-p_1-s_0-p_2-p_3-s_4-p_4-p_5-s_8-p_6-p_7$
23	$p_0-p_1-s_1-p_2-p_3-s_4-p_4-p_5-s_8-p_6-p_7$
24	$p_0-p_1-s_2-p_2-p_3-s_4-p_4-p_5-s_8-p_6-p_7$

3.7　条件覆盖

3.7.1　简单条件覆盖

一个简单判定是由一个简单条件语句所构成，那么一个判定就是一个条件判断语句。而在实际的应用中，一个判定往往是由多个条件组合而成。组合两个条件的逻辑运算符是：逻辑与、逻辑或。其逻辑与真值表如表3-8所示。

当逻辑与的结果为True时，可以推断出条件1和条件2的值均为True。在逻辑与的结果为False时无法判断条件1和条件2的值是True还是False。例如，在Python语言中的一个判定：a>1 and b==0。就是由a>1和b==0两个条件通过逻辑与运算所构成，当a<=1或者b!=0都有可能使得结果为False。在Python交互环境IDLE中，

很容易验证，如图 3-28 所示。

表 3-8 逻辑与的真值表

条件表达式 1	条件表达式 2	逻辑与
True	False	False
False	True	False
False	False	False
True	True	True

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>a=0
>>>b=1
>>>a>1 and b==0
False
>>>a=3
>>>b=1
>>>a>1 and b==0
False
>>>a=0
>>>b=0
>>>a>1 and b==0
False
>>>
```

图 3-28 IDLE 中的逻辑与验证

逻辑或真值表如表 3-9 所示。

表 3-9 逻辑或的真值表

条件 1	条件 2	逻辑或	条件 1	条件 2	逻辑或
True	False	True	True	True	True
False	True	True	False	False	False

在逻辑或中，只有判定结果为 False 时能够确定条件 1 和条件 2 的表达式均为 False，在结果为 True 时，无法判定哪个值为 True。例如，在 Python 语言中的一个判定 a>1 or b==0 就是由 a>1 和 b==0 两个条件通过逻辑或运算所构成，当 a>1 或者 b==0 时都有可能使得结果为 True。在 Python 交互环境 IDLE 中，很容易验证，如图 3-29 所示。

从上面的分析看出，分支覆盖并不能反映出判定中不同条件的情况，为了区分不同条件情况，引入条件覆盖：构造一组测试用例，使得每一判定语句中每个逻辑条件的可能值至少满足一次。

程序 3-28 存在两个判定，判定 1 由 x>2 以及 z<8 两个条件由逻辑与运算连接而成，第 2 个判定由 x==4 以及 y>6 两个条件通过逻辑或运算连接而成。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>a=2
>>>b=1
>>>a>1 or b==0
True
>>>a=0
>>>b=0
>>>a>1 or b==0
True
>>>a=2
>>>b=0
>>>a>1 or b==0
True
>>>a=0
>>>b=1
>>>a>1 or b==0
False
>>>
```

图 3-29　IDLE 中的逻辑或验证

程序 3-28　条件覆盖的程序

```
def work(x,y,z):
    k=0;
    j=0;
    if (x>2 and z<8):  判定 1
        k=x* y;
        j=k+z;
    if (x==4and y>6):  判定 2
        j=x* y+5;
    j=j% 4;
    return j;
```

如果用 T_1 表示 x>2 结果为真，F_1 表示 x>2 结果为假。用 T_2 表示 z<8 结果为真，F_2 表示 z<8 结果为假。T_3 表示 x==4 结果为真，F_3 表示 x==4 结果为假。用 T_4 表示 y>6 结果为真，F_4 表示 y>6 结果为假。设计测试用例，确保这 8 个条件值均出现一次就满足条件覆盖准则。

程序 3-29 给出了和程序 3-28 对应的满足条件覆盖的测试程序。

程序 3-29　和程序 3-28 对应的满足条件覆盖的测试程序

```
#test_switch1.py
import  pytest
from app.condition import calc

@pytest.mark.parametrize("x,y,z,r", [
    (3,8,7,3),
    (1,5,9,0),
```

```
    (4,5,8,0),
])
def test_demo1(x,y,z,r):
    assert calc(x,y,z)==r;
```

在上述测试数据中，测试用例(3,8,7,3)实现了 $T_1T_2F_3T_4$ 的覆盖，而(1,5,9,0)实现了 $F_1F_2F_3F_4$ 的覆盖。检查条件覆盖情况，T_3 并没有被覆盖，需要补充覆盖 T_3 的测试用例。这个测试用例具有(4,—,—,—)的形式，也就是 x=4，其他变量的取值不做限定，可以根据其他信息作为启发进行补充，现在取(4,5,8,0)作为该测试用例。

进一步观察可以发现，判定点 1 中的 x>2 和判定点 2 中的 x==4，存在使其条件结果重叠的情况，在设计测试用例时，可以用同样的数据来覆盖更多的条件。即取 x=4，那么 x>2 以及 x==4 均为 True，当 x=1 时，那么 x>2 以及 x==4 均为 False。对程序 3-29 所给出的测试用例进一步精简，如程序 3-30 所示。在这个测试程序中，第一个测试用例(4,8,7,1)实现了 $T_1T_2T_3T_4$ 的覆盖，而第二个测试用例实现了 $F_1F_2F_3F_4$ 的覆盖。

程序 3-30　满足条件覆盖的另一测试程序

```
#test_switch1.py
import  pytest
from app.condition import calc

@pytest.mark.parametrize("x,y,z,r", [
    (4,8,7,1),
    (1,5,9,0),
])
def test_demo1(x,y,z,r):
    assert calc(x,y,z)==r;
```

3.7.2　条件判定覆盖

在程序 3-28 中，满足条件覆盖的测试用例并不唯一，程序 3-29 和程序 3-30 都满足条件覆盖的准则。条件覆盖关注的是一个判定中的每一个条件的真假情况，而判定覆盖关注的是整个判定的最终结果。依据逻辑与、逻辑或的两个运算真值表，不同的组合可能产生同样的输出，满足条件覆盖不一定满足判定/分支覆盖准则。

程序 3-31 包含两路分支，并且其判定点由两个条件通过逻辑运算与连接而形成。

程序 3-31　包含简单与运算的判定

```
#import math
#E:\2014\bookpub\software_testing\python\pytestdemo6\app\condec.py
def condec(a,b):
    x=12;
    if (a==0 and b>2):
        x=x/a;
```

```
    else:
        x=x+b;
    x=x+1;
    return x;
```

和前面类似，判定点中的两个条件分别存在结果为 True 和 False 两种情况，只要构建的测试用例集分别使得两个条件分别取得结果为 True 和 False，就满足条件覆盖准则。测试程序 3-32 所包含的测试数据就可以满足上述条件。

程序 3-32　带有逻辑或判定中满足条件覆盖的测试

```
#test_condec1.py
import  pytest
from app.condec1 import condec;

@pytest.mark.parametrize("a,b,r", [
    (3,4,17),
    (0,1,14),
])
def test_demo1(a,b,r):
    assert condec(a,b)==r;
```

在这个测试程序中包含两个测试用例，分别为(3,4,17)和(0,1,14)。其中，(3,4,17)覆盖了 T_1F_2，而(0,1,14)覆盖了 F_1T_2。判定所有的条件都已经被覆盖，但是判定的结果都是 False，该判定在两个测试用例下都执行了判定结果为 False 的输出分支，如图 3-30 中的虚线所示，即(1)—(2)—(5)—(6)。显然这两个测试并没有使得判定为 True 的分支被覆盖到，不能满足判定/分支覆盖准则。由此可以知道，满足了条件覆盖准则并不能保证满足判定/分支覆盖准则，两者之间不具备包含关系。如有语句处于未被覆盖的分支上，那么该语句也不能够被执行，在图 3-30 中，x＝x/a 这条语句没有被测试所覆盖到，事实上在 a＝0 时恰恰在该语句存在被零除的错误，但是该错误未能被发现。

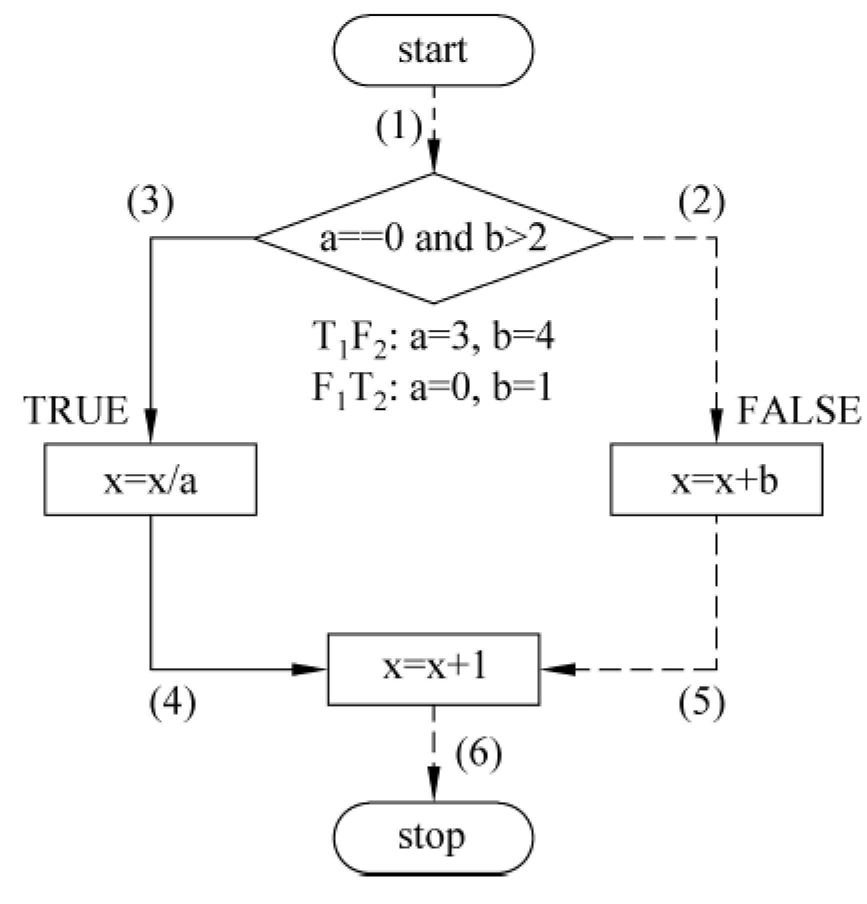


图 3-30　逻辑与运算下条件和分支的差异

在利用条件覆盖测试逻辑或所构成的判定时也有类似的情况。程序 3-33 给出了一个简单的仅包含逻辑或运算判定的示例程序，该程序包含两路分支，程序判定点由两个条件通过逻辑运算或连接而形成。

程序 3-33　包含简单逻辑或运算的程序

```
#import math
#E:\2014\bookpub\software_testing\python\pytestdemo7\app\condec2.py
def condec(a,b):
```

```
    x=12;
    if (a>2 or b!=0):
        x=x+a;
    else:
        x=x/b;
    x=x+1;
    return x;
```

和前面类似,判定点中的两个条件分别存在结果为 True 和 False 两种情况,只要构建的测试用例集分别使得两个条件分别取得结果为 True 和 False,就满足条件覆盖准则。程序 3-34 包含的测试数据就可以满足上述条件。

程序 3-34　带有逻辑或判定中满足条件覆盖的测试

```
#test_switch1.py
import  pytest
from app.condec import condec;

@pytest.mark.parametrize("a,b,r", [
    (3,0,16),
    (1,1,14),
])
def test_demo1(a,b,r):
    assert condec(a,b)==r;
```

这个测试程序包含两个测试用例,分别为(3,0,16)和(1,1,14)。其中,(3,0,16)覆盖了 T_1F_2,而(1,1,14)覆盖了 F_1T_2。判定所有的条件都已经被覆盖,但是判定的结果都是 True,该判定在两个测试用例下都执行了判定结果为 True 的输出分支,如图 3-31 中的虚线所示,即(1)—(2)—(4)—(6)。显然这两个测试并没有使得判定为 False 的分支被覆盖到,不能满足判定/分支覆盖准则。由此可以知道,满足了条件覆盖准则并不能保证满足判定/分支覆盖准则,两者之间不具备包含关系。如有语句处于未被覆盖的分支上,那么该语句也不能够被执行。例如,语句 x=x/b 没有被覆盖到。实际上,在 b=0 时恰恰在该语句存在被零除的错误,但是该错误未能被发现。

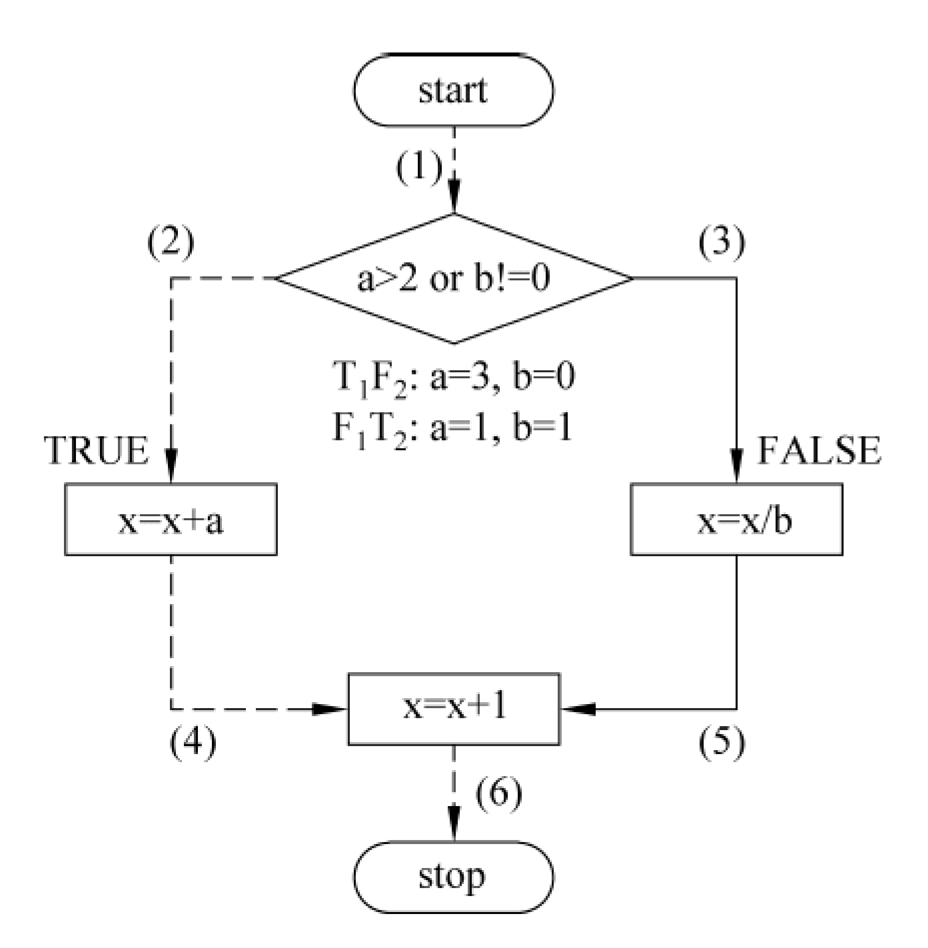


图 3-31　逻辑与运算下条件和分支的差异

无论是逻辑与还是逻辑或所构成的判定,都存在满足条件但是不满足判定覆盖的情况,同样处于未被覆盖分支上的语句也无法被覆盖到。因此,条件覆盖准则也并不能保证语句覆盖。因此,提出了条件判定覆盖:设计足够的测试用例,使得判定中每个条件的所有可能(真/假)至少出现一次,并且每个判定

本身的判定结果(真/假)也至少出现一次。

对于程序 3-31,测试程序 3-32 给出测试用例集,如表 3-10 中第一行和第二行所示,测试用例 1 和测试用例 2 仅覆盖了判定结果为 False 的分支,所以在此基础上,必须补充其判定结果为 True 的测试数据。对于变量 a 只能取值 0,而变量 b 取任意大于 2 的数值都可以,可以根据启发式信息做出抉择,这里将其取值为 4,其产生的判定结果为 True。显然无论在条件层次上还是在判定层次上均满足覆盖准则。

表 3-10　程序 3-31 中补充用例满足条件判定覆盖

编号	测试输入	覆盖的条件(逻辑与)		覆盖的判定
		a==0	b>2	
1	(3,4)	False	True	False
2	(0,1)	True	False	False
3	(0,4)	True	True	True

对于程序 3-33,测试程序 3-34 给出测试用例集,如表 3-11 中第一行和第二行所示,测试用例 1 和测试用例 2 仅覆盖了判定结果为 True 的分支,所以在此基础上,必须补充其判定结果为 False 的测试数据。对于变量 a 任意小于 2 的数值都可以,可以根据启发式信息做出抉择,这里将其取值为 1,变量 b 取 0,其产生的判定结果为 True。显然无论在条件层次上还是在判定层次上均满足覆盖准则。

表 3-11　程序 3-31 中补充用例满足条件判定覆盖

编号	测试输入	覆盖的条件(逻辑或)		覆盖的判定
		a>2	b!=0	
1	(3,0)	True	False	True
2	(1,1)	False	True	True
3	(1,0)	False	False	False

而实际上只由一个运算符构成的判定,无论与运算还是或运算,一个简单的构造满足条件判定准则的输入方法是让两个条件的结果同时为 True 或者同时为 False,其判定也必定包含 True 和 False 的结果。此时满足条件判定准则。当有多个条件构成,可以采用递归方法做进一步的分解,确定每一个输入参数的取值。

条件判定覆盖可以发现但是无法保证发现以下所有的缺陷。

(1) 逻辑变量错误。

(2) 逻辑运算括弧错误。

(3) 关系操作符错误。

(4) 算术表达式错误。

(5) 遗漏逻辑操作符。

(6) 多余逻辑操作符。

(7) 不正确逻辑操作符。

3.7.3 条件组合覆盖

在条件组合中，将所有逻辑变量的取值为 True 和 False 的两种情况全部组合一次。显然若一个判定具有 n 个独立的条件，那么最后的组合数为 2^n。显然随着 n 的增大，测试用例数将以指数方式增长。由于用例较多，本节直接利用表格表示测试用例。

例如，判定 R=(a<5 and b<10 or c=20 and d>30)，a、b、d 的取值在原则上由其他启发式信息决定，例如边界值、等价类等。假设变量 a 小于 5 时取值 4，大于 5 时取值 6。b 小于 10 时，取值 9，大于 10 时取值 11。c 不等于 20 时取值 15，当 d 小于 30 时取值 29，大于 30 时取值 31。产生的测试用例，如表 3-12 所示。

表 3-12　条件组合覆盖示例

编号	测试用例				覆盖条件			
	a	b	c	d				
1	4	9	20	31	T_1	T_2	T_3	T_4
2	4	9	20	29	T_1	T_2	T_3	F_4
3	4	9	15	31	T_1	T_2	F_3	T_4
4	4	9	15	29	T_1	T_2	F_3	F_4
5	4	11	20	31	T_1	F_2	T_3	T_4
6	4	11	20	29	T_1	F_2	T_3	F_4
7	4	11	15	31	T_1	F_2	F_3	T_4
8	4	11	15	29	T_1	F_2	F_3	F_4
9	6	9	20	31	F_1	T_2	T_3	T_4
10	6	9	20	29	F_1	T_2	T_3	F_4
11	6	9	15	31	F_1	T_2	F_3	T_4
12	6	9	15	29	F_1	T_2	F_3	F_4
13	6	11	20	31	F_1	F_2	T_3	T_4
14	6	11	20	29	F_1	F_2	T_3	F_4
15	6	11	15	31	F_1	F_2	F_3	T_4
16	6	11	15	29	F_1	F_2	F_3	F_4

在上述讨论过程中，4 个条件是完全独立的，一个条件的取值结果不会对另一个条件结果产生任何的影响。如果不同条件之间的关系不是完全独立的，那么在组合过程中有些用例就不可能出现。例如，判定 R=a<5 and a<10，and 运算符前后两个条件并不完全独立。当 a<5 结果为 True 时，a>10 的条件只有一种取值 False。

组合条件的优点是：测试全面。

其缺点是：测试用例数量大，当出现问题时定位比较困难。

3.8 修正条件判定覆盖

3.8.1 修正条件判定覆盖的定义

在 Python 语言中，由于存在短路计算现象，导致很多缺陷无法发现。关于短路计算前面已经进行过说明，这里仅简要分析。程序 3-35 给出了一个例子。

程序 3-35 一个具有短路效应的判定

```
def a():
    print 'this is A!'
    return 1
def b():
    print 'this is B!'
    return 1
def c():
    print 'this is C!'
    return 1
if a() or b() and not c():
    print 'OK!'
```

这个代码段执行的结果为：

```
>>>
this is A!
OK!
>>>
```

该程序段中的判定：

```
a() or b() and not c()
```

实际上等效于：

```
a() or (b() and not c())
```

尽管 and 运算以及 not 运算的优先级均高于 or 运算，但是由于最后的 or 运算只要左边的返回值能够直接确定整个判定的值，那么其右边的值将不再运算。and 的优先级高于 or 的含义是在计算 or 运算之前先算其左边的 a()值，然后是右边 and 运算，最后执行 or 运算。而当左边能够确定整个判定值的情况下，右边不再计算。换句话说，Python 并非简单地根据优先级执行运算。在这种情况下，可能存在一些缺陷无法被发现，例如 c()或者 b()函数中存在的缺陷。

为了描述判定的输入和输出的关系，特别是为了避免产生计算短路现象，业界提出了修正的条件判定覆盖。这个度量最早是波音公司创建的，并被用于航空软件标准 RCTA/DO-178B 中。航空电子标准 RTCA/DO-178B 的 A 级认证要求程序的每一行代

码都要进行 MC/DC 覆盖测试。

修正条件判定覆盖(Modified Condition/Decision Coverage,MC/DC)方法要求程序中的每一个条件必须产生所有可能的输出结果至少一次,并且每一个判定中的每一个条件必须能够独立影响一个判定的输出,即在其他条件不变的前提下仅改变这个条件的值,而使判定结果改变。修正的条件判定覆盖,包含以下三个含义。

(1) 判定的每个条件所有取值至少出现一次。

(2) 每个判定的所有可能结果至少出现一次。

(3) 每个条件都能独立地影响判定的结果。

MC/DC 发现的主要软件问题包括以下几种。

(1) ORF:Operator Reference Faults,例如"与"被误写成"或"。

(2) VNF:Variable Negation Faults,一个变量被误写成了它的否定。

(3) ENF:Expression Negation Faults,一个表达式被误写成了它的否定。

一般而言,一个判定是由逻辑运算符(主要是与运算、或运算)连接而构成。为了分析复杂的判定的 MC/DC 的设计方法,先观察两个最简单的运算规律。以 R=A and B 为例进行讨论。列出条件 A 和条件 B 的取值,如表 3-13 所示。

表 3-13　A and B 的 MC/DC 覆盖

测试用例编号	A	B	R=A and B
1	True	True	True
2	False	True	False
3	True	False	False

在测试用例 1 和测试用例 2 中,在 B 为 True 的前提下,结果 R 的值随着 A 的变化而变化,即 A 独立地影响判定结果。而在测试用例 1 和测试用例 3 中,A 为 True,R 的值随着 B 的变化而变化,即 B 独立地影响了结果 R。显然表 3-13 满足 MC/DC 的覆盖要求。

同理,可以列出 R=A or B 满足 MC/DC 覆盖的测试用例,如表 3-14 所示。

表 3-14　A or B 的 MC/DC 覆盖

测试用例编号	A	B	R=A or B
1	False	False	False
2	False	True	True
3	True	False	True

在测试用例 1 和测试用例 2 中,在 A 为 False 的前提下,结果 R 的值随着 B 的变化而变化,即 B 独立地影响判定结果。而在测试用例 1 和测试用例 3 中,B 为 False,R 的值随着 A 的变化而变化,即 A 独立地影响了结果 R。显然表 3-13 满足 MC/DC 的覆盖要求。

若一个判定具有 $n(n\geqslant 2)$ 个逻辑变量(条件),则满足 MC/DC 覆盖要求的测试用例至少需要 $n+1$ 个测试用例。

证明：利用归纳法。

当 $n=2$ 时，根据表 3-13 和表 3-14，显然至少三个测试用例才能满足 MC/DC 覆盖条件。

假设当 $n=j$ 时，至少需要 $j+1$ 个测试用例，当 $n=k$ 时，至少需要 $k+1$ 个测试用例。

现在证明当 $n=j+k$ 时，至少需要 $j+k+1$ 个测试用例才能满足 MC/DC 覆盖。

先假设通过 and 运算符连接两个子判定：

$$R=D_1 \text{ and } D_2$$

其中，D_1 具有 j 个原子逻辑变量（条件），D_2 具有 k 个原子逻辑变量（条件）。其中，D_1 中满足 MC/DC 的测试用例集表示为 $\{S_{11},S_{12},\cdots,S_{1j+1}\}$，而 D_2 中满足 MC/DC 的测试用例集表示为 $\{S_{21},S_{22},\cdots,S_{2k+1}\}$。

显然在 $\{S_{11},S_{12},\cdots,S_{1j+1}\}$ 中至少包含一个使得 D_1 为 True 的测试用例，不妨假设为 S_{11}。同样 $\{S_{21},S_{22},\cdots,S_{2k+1}\}$ 至少包含一个使得 D_2 为 True 的测试用例，不妨假设为 S_{21}。现在，利用 $\{S_{11},S_{12},\cdots,S_{1j+1}\}$ 和 $\{S_{21},S_{22},\cdots,S_{2k+1}\}$ 构建的 R 是满足 MC/DC 的测试用例集。

为了使得 D_1 的变量能够独立影响结果 R，那么 D_2 的结果必须为 True。选择 D_2 表示的测试用例为 S_{21}，并且将其与 $\{S_{11},S_{12},\cdots,S_{1j+1}\}$ 组合构成新的测试用例集：

$$TS_1=\{S_{11}\ S_{21},S_{12}\ S_{21},\cdots,S_{1j+1}\ S_{21}\}$$

测试用例集 TS_1 必然满足使得 D_1 的每一个条件独立影响 R 的结果。

同理，为了使得 D_2 中原有逻辑变量能够独立影响结果 R，那么 D_1 的结果必须为 True，选择 D_1 的测试用例 S_{11}，并且将其与 $\{S_{21},S_{22},\cdots,S_{2k+1}\}$ 组合构成新的用例：

$$TS_2=\{S_{11}S_{21},\ S_{11}S_{22},\cdots,\ S_{11}S_{2k+1}\}$$

测试用例集 TS_2 必然满足使得 D_2 的每一个条件独立影响 R 的结果。

考虑 D_1 和 D_2 中的所有逻辑变量，并且使其满足 MC/DC 覆盖的测试用例集：

$$\begin{aligned}TC&=TC_1\cup TC_2\\&=\{S_{11}\ S_{21},S_{12}\ S_{21},\cdots,S_{1j+1}\ S_{21},\ S_{11}S_{22},\cdots,\ S_{11}S_{2k+1}\}\end{aligned}$$

根据前面的假设，TS_1 和 TS_2 都是最小测试用例集，无法最精简。并且 TS_1 和 TS_2 只有一个共同测试用例 $S_{11}S_{21}$，所以 TC 的元素最小个数为：

$$(j+1)+(k+1)-1=(j+k)+1$$

同理，可以证明当：

$$R=D_1 \text{ or } D_2$$

其满足 MC/DC 的最小测试用例个数也是 $(j+k)+1$。

证毕。

MC/DC 由于本身的优点，在工业中得到了非常广泛的应用。但是生成满足 MC/DC 条件的测试用例并不是一件非常容易的事，学者对 MC/DC 的测试用例生成方法进行了深入的研究。目前常用的 MC/DC 的设计方法包括唯一原因法、屏蔽法、二叉树法等。

3.8.2 唯一原因法生成 MC/DC 测试用例

唯一原因法是一种较早的 MC/DC 测试用例设计方法，通过条件的真值表比对找出

在其他条件不变的前提下影响输出的条件。包括以下三个基本步骤。

(1) 列出条件的全组合。

(2) 找出独立影响对。

(3) 找出最小测试用例集。

下面以 R=A and (B or C) 为例说明唯一原因法的原理,第一步列出 A、B、C 三个条件所有组合并计算输出值,如表 3-15 中的左边 4 列所示。

表 3-15　A and(B or C)输入和输出的关系

测试用例编号	A	B	C	R	独立影响		
					A	B	C
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

在表 3-15 中,比对不同的行,找出独立影响输出的测试用例。例如,在测试用例 1 和测试用例 5 中,输入分别为 TTT 和 FTT,输出为 T 和 F,如表中阴影部分所示。显然在 B 和 C 固定为 TT 的前提下,A 独立影响了输出结果 R。在表格右边的第一行 A 所对应的位置填上 5,而在第五行 A 所对应的位置填上 1。其含义是测试用例 1 和测试用例 5 可以使得条件 A 独立影响判定输出 R。同理,测试用例 2 和测试用例 4,在 A 和 C 固定为 TF 的前提下,可以使得条件 B 独立地影响判定输出 R。以此类推,完成右边的表格。

在上述的基础上,将每一个条件能够独立影响的输出的测试用例对重新整理,形成表 3-16。

表 3-16　A and(B or C)独立影响判定的条件测试用例对

条件	测试用例对 1	测试用例对 2	测试用例对 3
A	5,1	6,2	3,7
B	2,4	—	—
C	3,4	—	—

根据表 3-16 找出最小的测试用例集。对于条件 B 和条件 C 都只有一个测试用例对能够满足独立影响条件,所以这两个测试用例对都必须作为测试用例。由于在这两个用例对中间,测试用例 4 是重复的,所有只要使用一次即可。可以将测试用例 TS={2,3,4}添加到最后测试集中。条件 A 可以在三个候选测试用例中选择。测试用例对 2 和测试

用例对3和已经选择测试用例TS={2,3,4}都有一个用例是重复的,而测试用例对1和已经选择的TS没有任何的重复。所以在测试用例对2或者测试用例对3中选择。若选择的是测试用例对2,那么形成的测试用例集TS={2,3,4,6},如表3-17所示。

表3-17 A and(B or C)满足MC/DC的测试用例

测试用例编号	A	B	C	R
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
6	F	T	F	F

3.8.3 屏蔽法生成MC/DC测试用例

大部分的多目条件判定都认为是由简单的逻辑与或者逻辑或连接而形成,非运算是单目运算。根据表3-13和表3-14的分析,先隐藏一个运算对结果的影响。对于R= A and(B or C),为了确认A能够对于R存在对立影响,那么B or C的结果必须为True。此时条件B和条件C有一个为True即可。选择B=True,C=False,那么可以形成两个测试用例,如表3-18所示。

表3-18 A and(B or C)中A对于R的独立影响

测试用例编号	A	B	C	R
1	T	T	F	T
2	F	T	F	F

同样,为了讨论条件B对结果的影响,A只能选择True,而C只能选择False。形成两个测试用例,如表3-19所示。

表3-19 A and(B or C)中B对于R的独立影响

测试用例编号	A	B	C	R
3	T	T	F	T
4	T	F	F	F

同样,为了讨论条件C对结果的影响,A只能选择True,而B只能选择False。形成两个测试用例,如表3-20所示。

表3-20 A and(B or C)中C对于R的独立影响

测试用例编号	A	B	C	R
5	T	F	T	T
6	T	F	F	F

在上述测试用1到测试用例6中,测试用例1和测试用例3是重复的,测试用例4和测试用例6是重复的,重新整理得到最后的测试用例集如表3-21所示。

表3-21 完成的测试用例

测试用例编号	A	B	C	R
1	T	T	F	T
2	F	T	F	F
3	T	F	F	F
4	T	F	T	T

3.8.4 二叉树法生成MC/DC测试用例

在二叉树法中,首先必须将判定表达式表示成为一个判定二叉树。在该判定二叉树中,中间节点为二目运算符(这里只考虑逻辑与、逻辑或,不考虑非运算以及异或运算,非运算为单目运算,只要将其逻辑取反即可,而逻辑异或可以表示成为与运算和或运算的组合),而叶子节点是一个判定中的单一条件。不同优先级的运算符通过二叉树的层次来表示,位于树根的运算法优先级最低,越接近叶子节点的运算符优先级越高。

一个判定表达式包含and、or、(、)4个字符。在没有括号的情况下,and的优先级高于or,相邻的两个相同的运算符优先级从左到右。如果有括号,那么括号内运算优先。构建判定二叉树的方法,和执行中缀表达式的算法类似。构建两个堆栈,分别为运算符堆栈和逻辑变量堆栈。从左到右扫描判定表达式,如果是变量表达式则压入变量堆栈,若为运算符,则将其和运算符堆栈的栈顶运算符比较,若栈顶运算符的优先级高,则将栈顶的运算符弹出,从变量堆栈弹出两个变量,分别作为该运算符的左右孩子,加入变量栈。若为左括号则将其入栈,若是右括号,则不断执行动作(从运算符堆栈中弹出运算符,同时从变量运算符堆栈弹出两个变量,分别作为该运算符的左右孩子,加入到变量栈),一直到遇到左括号。若扫描完毕,运算符堆栈不为空,值持续执行动作(从运算符堆栈中弹出运算符,同时从变量运算符堆栈弹出两个变量,分别作为该运算符的左右孩子,加入到变量栈)。若运算符堆栈为空,则其变量栈中保存的就是判定二叉树。

判定R=(A and B)and(C or(D and E))对应的判定二叉树如图3-32所示。

现规定根节点的层数为1,根节点的左右孩子的层数为2。同时,对于每一个节点进行了编号,标号标注在节点的左边。对于一个逻辑变量,变量到根节点所经过节点构成的路径,称为变量的影响路径。在图3-32(a)中的判定二叉树,变量A的影响路径由节点4,2,1所构成,而变量D的影响路径由节点8,7,3,1所构成。

二叉判定树的方法构建满足MC/DC覆盖的测试用例集,选定独立影响判定结果的变量,除了该变量到树根节点所经过的独立影响路径外,通过两轮标记确定各个逻辑变量的值。对于一个逻辑变量,具体包括以下几个步骤。

(1) 选定变量。选定独立影响判定结果的变量,也就是判定二叉树的一个叶子节点。

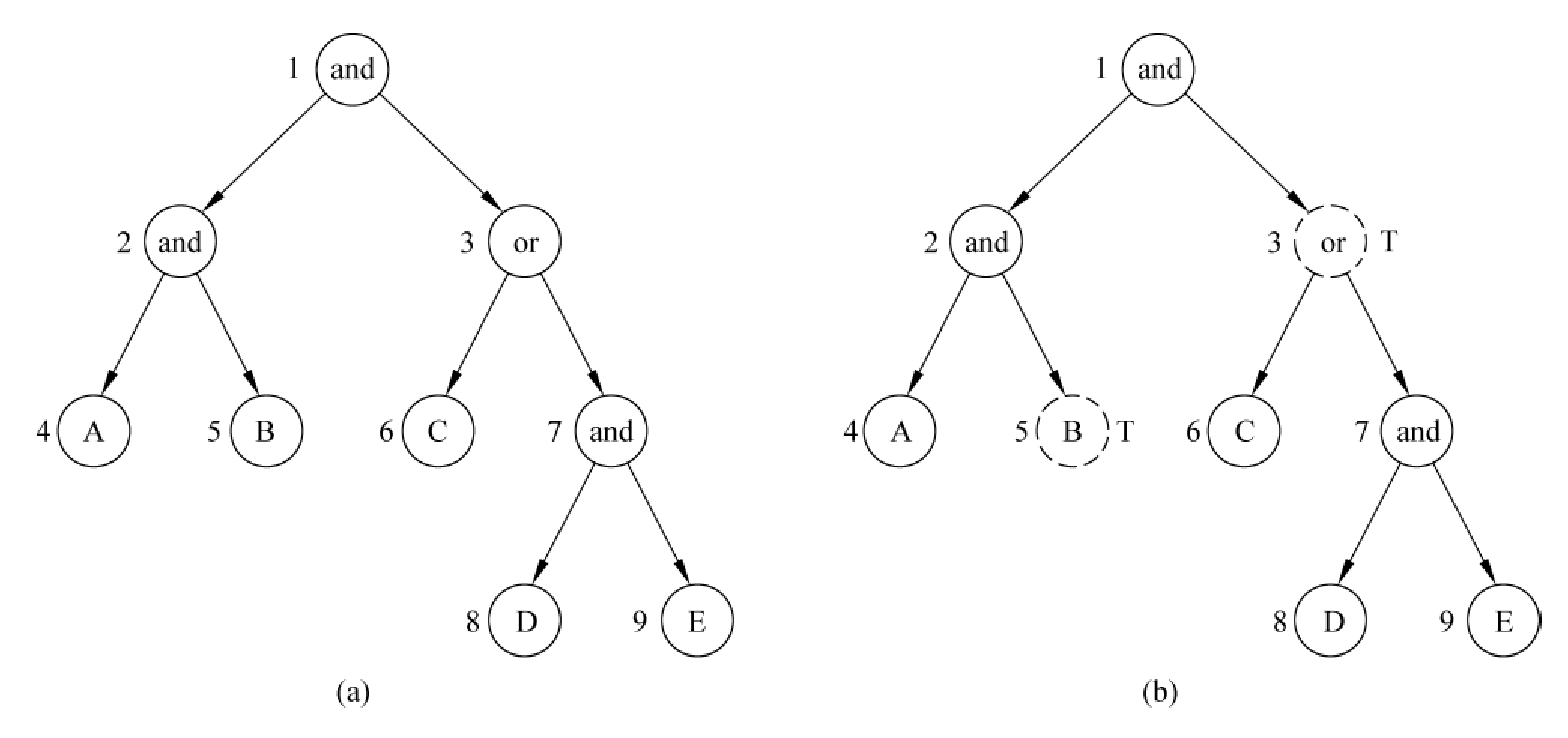


图 3-32 判定二叉树及其向上标注

(2) 向上标注。从该节点开始向上访问每一个节点,根据其父节点的运算符确定其兄弟节点的逻辑值。若父节点为 and 运算符,则其兄弟节点标记为 True。若其父节点为 or 运算符,则其兄弟节点标记为 False 运算符。如此反复,一直到父节点为根节点。

(3) 向下标注。在判定树的第二层,从不在独立影响路径的节点开始,向下访问一个节点,标记节点的逻辑值。若该节点的标记值为 True,而节点的运算符为 and,则其两个孩子节点均标记为 True。若该节点标记值为 False,而节点运算符为 and,则两个孩子节点任意选择一个为 False,另外一个值可以为 True 或者 False。若该节点的标记值为 False,而节点的运算符为 or,则其两个孩子节点均标记为 False。若该节点的标记值为 True,而节点的运算符为 or,则其一个孩子标记为 True,另外一个孩子的值可以为 True 或者 False。

(4) 构造用例。对于选定的变量的两个取值 True 和 False,和其他所有叶子节点通过上述(2)和(3)步骤确定的值构成两个测试用例。

对于每一个逻辑变量,都采用上述方法确定其测试用例集。

下面以其中的逻辑变量 A 为例,来阐述上述的过程。

向上标注:变量 A 所在的节点编号为 4,其父节点为节点 2,包含运算符 and,其兄弟节点为节点 5。根据第二条规则,将 B 的值设为 True。接下来,考虑第二层的节点,节点 2 位于影响路径上,其父节点的内容为 and,所以可以将其兄弟节点设置为 True。由于节点 2 的父节点为根节点,自此,完成向上的标记过程,如图 3-32(b)中的虚框所示。

向下标注:从第二层开始,搜索影响路径以外的所有没有标注的节点。节点 2 处于影响路径上,而节点的两个孩子分别为节点 6 和节点 7,由于节点自身的运算符为 or,而节点标注的结果为 True,所以对于节点 6 和节点 7 至少需要一个标注为 True,现在选择将两个节点均标注为 True,如图 3-33(a) 所示。接下来看,节点 7 的左右孩子都没有标注,其本身已经标注为 True,所以其左右孩子均标注为 True。在这个过程中,标注的结果并不唯一。例如,节点 3 已经被标注为 T,而其运算符内容为 or,左右两个孩子如果节

点标注为 True,而节点 7 标注为 False,就会得到另一个测试用例集。

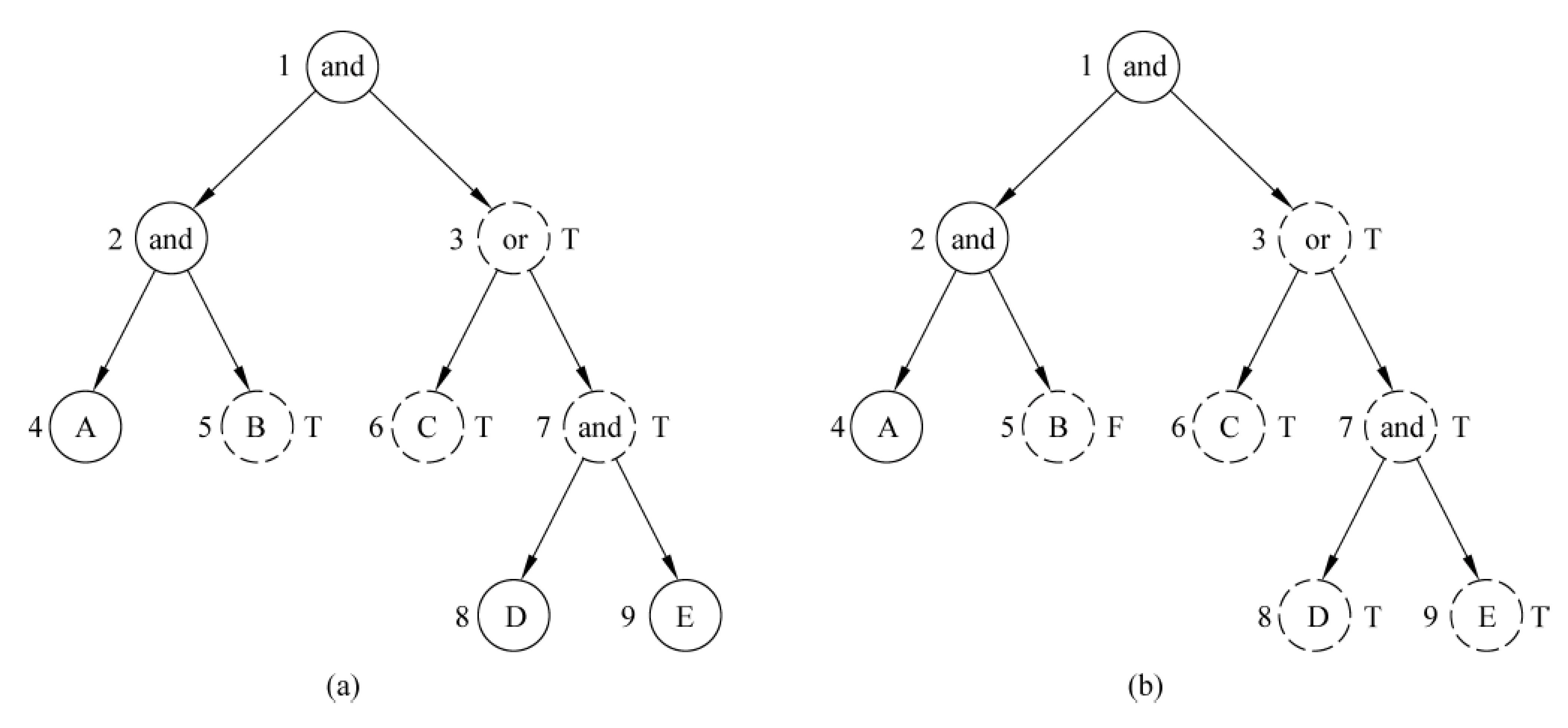


图 3-33　二叉判定树的向下标注

通过向上标注和向下标注,完成除影响路径以外的所有节点的标注。将完成标注的所有叶子节点组合,形成逻辑变量 A 的独立影响判定 R=(A and B) and(C or(D and E))的测试用例集,如表 3-22 所示。

表 3-22　二叉树法形成对变量 A 的独立影响测试用例集

编号	A	B	C	D	E	R
1	T	T	T	T	T	T
2	F	T	T	T	T	F

同理可以确定 B、C、D、E 变量的独立影响测试用例集,如表 3-23 所示,其中带底纹的格子表示独立影响判定结果。

表 3-23　二叉树法形成对变量 B、C、D、E 的独立影响测试用例集

编号	A	B	C	D	E	R
1	T	T	T	T	T	T
2	T	F	T	T	T	F
3	T	T	T	F	T	T
4	T	T	F	F	T	F
5	T	T	F	T	T	T
6	T	T	F	F	T	F
7	T	T	F	T	T	T
8	T	T	F	T	F	F

在利用二叉树法产生的这些测试用例集中,中间存在重复的测试用例,例如,表 3-22

中的测试用例 1 和表 3-23 中的测试用例 1、表 3-23 中的测试用例 4 和测试用例 6、测试用例 5 和测试用例 7，删除这些重复的测试用例，使其仅保留一个非重复的测试用例，最后形成完整的满足 MC/DC 覆盖的测试用例集，如表 3-24 所示。

表 3-24 二叉树法构造的满足 MC/DC 覆盖测试用例集

编号	A	B	C	D	E	R
1	T	T	T	T	T	T
2	F	T	T	T	T	F
3	T	F	T	T	T	F
4	T	T	T	F	T	T
5	T	T	F	F	T	F
6	T	T	F	T	T	T
7	T	T	F	T	F	F

3.8.5 MC/DC 的进一步讨论

上面讨论了关于 MC/DC 问题的生成的三种方法。在实际应用中，判定和条件之间的关系往往比较复杂，例如，逻辑变量的独立性问题、if 语句构成多路分支和多条件判定的等效性问题、间接条件嵌套问题等，这些问题都会影响到所设计的测试用例覆盖程度，下面就这些问题展开讨论。

（1）逻辑变量的独立性问题。在前面的讨论过程中，不同的逻辑变量假设是独立的，换句话说，一个逻辑变量值和另外一个变量没有任何的约束关系。但是在实际应用中，不同逻辑变量之间的取值存在一定的约束关系。例如，程序 3-36 中的判定 A and B 并不独立。当 A 为 True 时，B 的值必定为 False，而当 A 为 False 时，B 的值可能为 True 和 False 两种取值。A and B 的值相对 C 是独立的，可做一个变量替换 A_1 = A and B，那么判定可以看成为 A_1 or C，这时 A_1 和 C 是完全独立的。

程序 3-36 判定之间独立性示例

```
def fun(a,b):
    A=5<a
    B=a<10
    C=b>10
    if (A and B) or C:
        print "True"
    else:
        print "False"
```

（2）if 语句构成的多路分支和多条件判定的转换。由多个 if 语句构成，若每一个语句都只在同一个结果(True 或者 False)分支中，那么其等效于一个多条件判定。在图 3-34 中，

左边是由 if 语句构成的多路分支伪代码，其对应的判定如右边所示。在这种情况下，右边 MC/DC 覆盖和左边的分支覆盖是等效的。

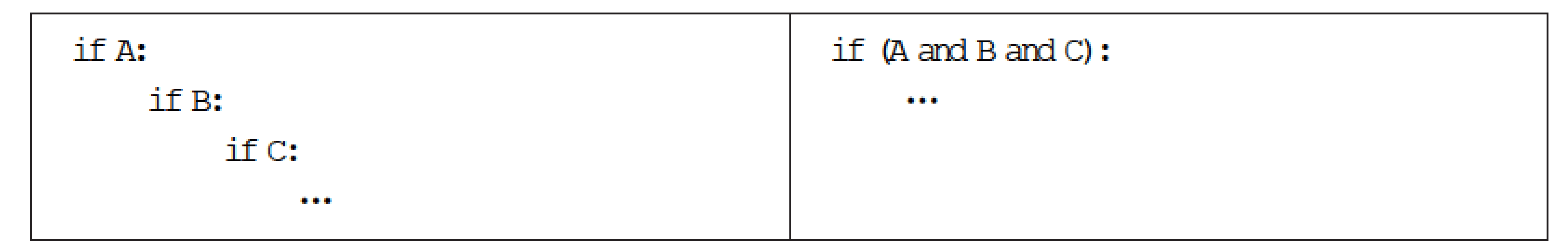


图 3-34　多路分支和多条件判定

（3）间接的条件嵌套问题。在前面讨论的过程中，均认为逻辑变量无法再拆分成更小的逻辑表达式。在实际的代码中，可能存在将一个复杂的逻辑表达式赋值给一个逻辑变量，然后再将该变量作为判定中的逻辑变量。如果仅考虑判定中的变量的直接 MC/DC 覆盖，将会遗漏一些发现缺陷的机会。

若有如下伪代码段：

```
A=A1 and B1
B=C1 or  D1
if(A and B):
    …
```

对于判定 R＝A and B，若仅考虑本判定自身，仅需要三个用例即可，如表 3-25 中的 A 列、B 列、R 列所示。但是对于 A 或者 B 的产生，并没有要求满足 MC/DC 覆盖。例如，当 A 为 F 时，可能由 A_1 和 B_1 的三种情况所构成，而 B 为 True 也可能由两种情况构成。在这几种情况下只要选取一种情况即可。当 $A_1B_1C_1D_1$ 取值分别为{TTTF，FTTT，TTFF}时构成的测试用例集就满足了 R＝A and B 的 MC/DC 的覆盖要求。

表 3-25　条件嵌套

<table>
<tr><th>编号</th><th>A₁</th><th>B₁</th><th>A</th><th>C₁</th><th>D₁</th><th>B</th><th>R</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">T</td><td rowspan="2">T</td><td rowspan="2">T</td><td>T</td><td>F</td><td rowspan="2">T</td><td rowspan="2">T</td></tr>
<tr><td>F</td><td>T</td></tr>
<tr><td rowspan="3">2</td><td>F</td><td>T</td><td rowspan="3">F</td><td>T</td><td>T</td><td rowspan="3">T</td><td rowspan="3">F</td></tr>
<tr><td>T</td><td>F</td><td rowspan="2">F</td><td rowspan="2">T</td></tr>
<tr><td>F</td><td>F</td></tr>
<tr><td>3</td><td>T</td><td>T</td><td>T</td><td>F</td><td>F</td><td>F</td><td>F</td></tr>
</table>

实际上，这段伪代码的实际效果等效于：

```
R=(A1 and B1)and(C1 or D1)
```

对于具有 4 个逻辑变量的判定表达式，若为了满足 MC/DC 覆盖，最少的测试用例数为 5 个。显然当 $A_1B_1C_1D_1$ 取值分别为{TTTF，FTTT，TTFF}时无法满足 MC/DC 的覆盖要求。为了避免造成的缺陷，在设计测试用例时，将 A 和 B 的表达式替换到 R 的表达式中，然后根据 MC/DC 准则进行测试用例的设计。

3.9 路径覆盖

无论是语句覆盖、判定覆盖、条件覆盖以及修正的条件判定覆盖，其关注点都是程序中的一个片段，并没有关注代码前后的逻辑关系。为了表示程序中前后的逻辑关系，发现中间可能存在的错误，必须分析程序基本控制流图，并测试控制流图中的路径。

3.9.1 程序和控制流图表示

不管业务逻辑如何复杂，所有的程序代码在结构上都是由顺序结构、分支结构、循环结构三种结构通过不同的方式组合而形成，3.3 节分别描述了这些结构的形式及其含义。由于在路径测试过程中，测试的关注点在于全局的路径，而不关心每一个节点的逻辑控制情况。为了突出全局性，需要将流程图转化为控制流图。

在控制流图中，有两个重要的组件：节点和有向弧(边)。

节点：用○表示控制流图的一个节点，它表示一个或者多个无分支的语句。为了讨论的方便，以及不同节点之间所构成路径的差异，在○中标记不同数字。

边(弧)：用有方向的弧线表示节点之间执行的逻辑次序，称为有向边(弧)。

一个控制流图必定具有如下特征。

(1) 具有一条入口边，该边没有起始节点，该边指向的节点称为入口节点。

(2) 具有一条出口边，该边没有终止节点，该边起始的节点称为出口节点。

(3) 所有的节点都至少有一条入边，至少有一条出边。

(4) 图中的任意一个节点，都在一条从入口边到出口边的路径上。

(5) 图中的任意一条边，都在一条从入口边到出口边的路径上。

路径是由节点构成的有序序列，该序列的初始节点为入口节点，而终止节点为出口节点，并且序列中两个相邻节点之间必定存在一个有向边。

和程序流程图不同，控制流图仅关注逻辑走向，而不关注中间的每一条语句的具体含义。在顺序结构中，尽管可能包含多条语句，但是在控制流图中仅用一个节点表示，如图 3-35 所示。

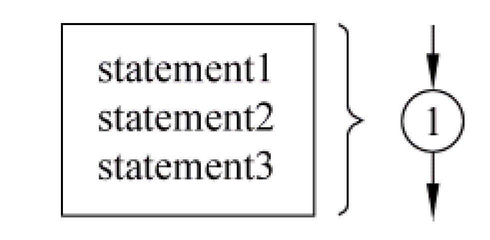


图 3-35 顺序结构的控制流图

图 3-36 给出了 if 以及 if-else 构成的控制流图示意图。在左边的 if 分支结构中，如果仅有 if 分支，statement1 是入口节点，当判定计算结果为 True 时执行一个语句体 statement2(可能是一条语句，也可能是多个顺序执行语句)，接着执行 statement3。而当判定结果为 False 时，直接跳过语句 statement2 执行 statement3。其可能的执行路径，用控制流图的节点表示如下。

路径 1：1—2—3—4

路径 2：1—2—4

在右边的 if-else 结构中，当 decision1 的结果为 True 或者 False 时，分别执行语句体 statement2 和语句体 statement3。其可能的执行路径，用控制流图的节点表示如下。

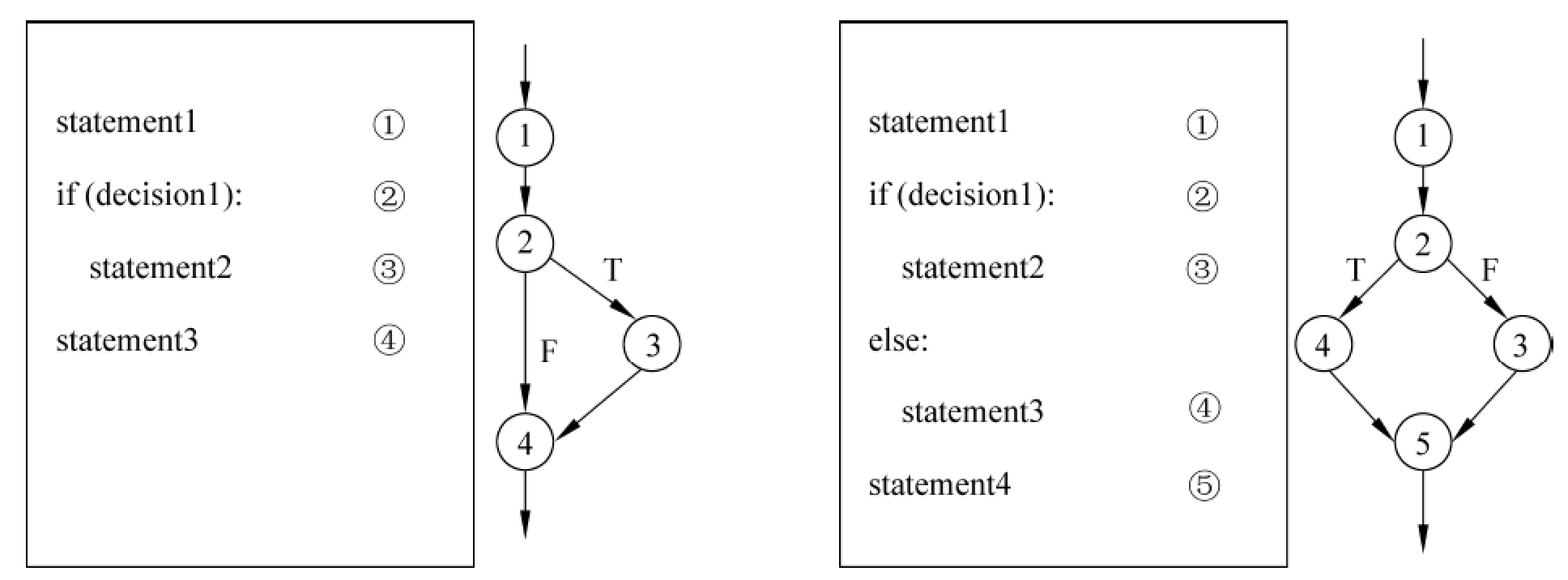


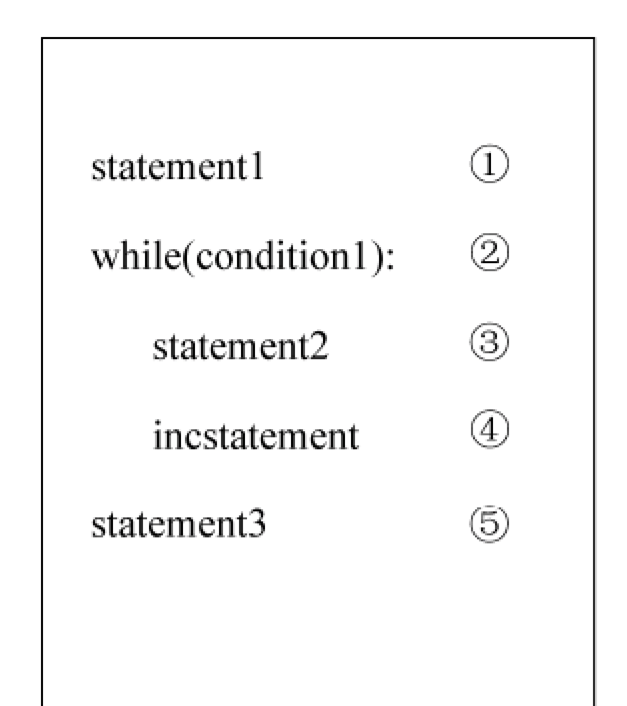
图 3-36 if 和 if-else 构成的控制流图

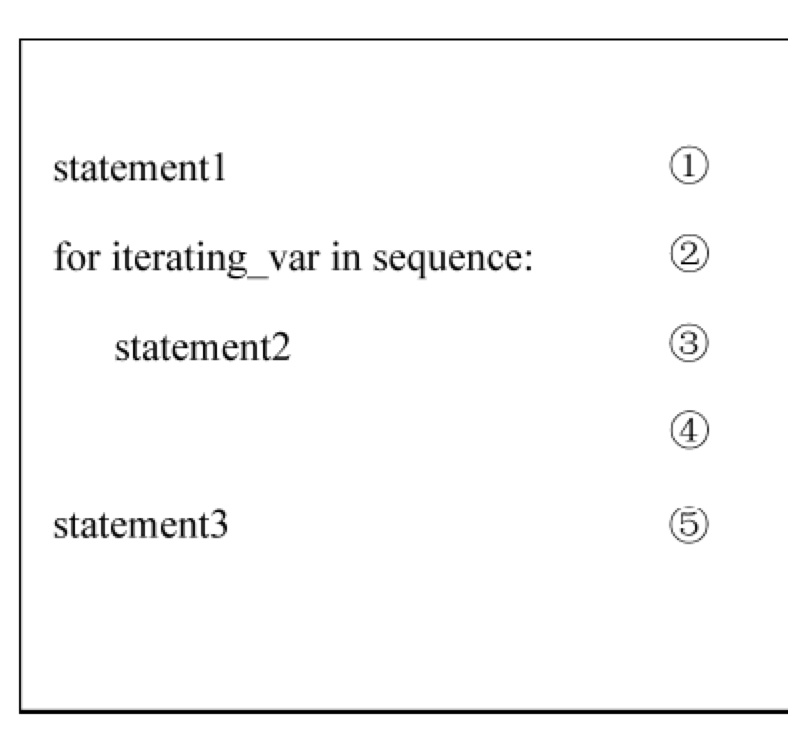
路径 1：1—2—3—5

路径 2：1—2—4—5

除了二路分支以外，还有 if-elseif 构成多路分支，其情况类似，不再重复。

除了 if 语句构成的路径以外，循环语句也是复杂控制流的重要组成要素。在 Python 中循环有两种形式：while 循环和 for 循环。在 Python 中没有 repeat 语句，实际上 repeat 语句是直接用 while 语句实现的。在 while 循环中，存在显式的循环变量的变更，图 3-37 右边的 incstatement 语句用于变更循环变量。在 for 循环中，循环控制通过访问可迭代对象来实现，并没有独立的循环变量变更语句。一个简单的例子是求 1～10 之间的累加，利用 while 和 for 两种不同的实现方式。在程序 3-37 中左边的是用 while 语句实现，显式地定义了循环变量 i，并且在循环体中，显式通过 i+=1 实现循环变量的变化。而右边循环变量的定义以及改变均包含在 for 语句内部。





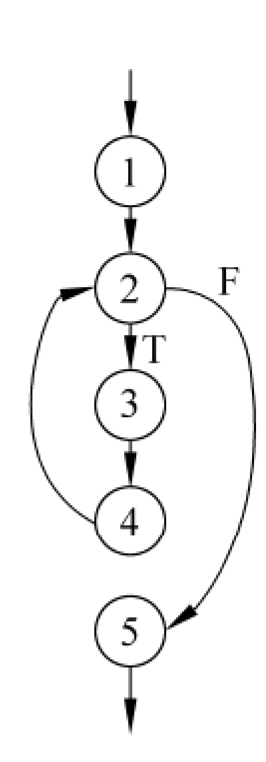


图 3-37 while 循环和 for 循环构成的控制流图

程序 3-37 while 和 for 实现的累加运算的等效性

```
def sum1():
    sum1=0
    i=1
    while(i<11):
        sum1=sum1+i
        i+=1
print sum1
```

```
def sum():
    sum=0
    for i in range(1,11):
        sum=sum+i
print sum
```

为了讨论的方便，增加逻辑的循环变量语句虚拟节点，如图 3-37 中的伪代码所示。在考虑了虚拟节点以后，while 循环变量和 for 循环的控制流程图就一致了，如图 3-37 右边所示。在这个控制流图中，入口节点到出口节点之间存在如下很多条路径。

路径 1：1—2—5

路径 2：1—2—3—4—2—5

路径 3：1—2—3—4—2—3—4—5

路径 4：1—2—3—4—2—3—4—2—3—4—5

……

其路径的条数是由循环变量所控制的。

在 Python 循环体中，有两个重要的影响程序控制流走向的语句，分别为 break 和 continue 语句。break 语句用来终止循环语句，即循环条件没有 False 条件或者序列还没被完全递归完，也会停止执行循环语句。在 while 和 for 循环中，都可以使用 break 语句。在使用嵌套循环时，break 语句将停止执行最深层的循环，并开始执行下一行代码。continue 语句跳出本次循环，而 break 跳出整个循环。continue 语句用来告诉 Python 跳过当前循环的剩余语句，然后继续进行下一轮循环。同样在 while 和 for 循环中，都可以使用 continue 语句。通常情况下，break 语句和 continue 语句都结合一个判定使用。图 3-38 给出了带有 break 和 continue 语句的控制流图。由于 break 和 continue 语句本身只控制流程方向，语句自身不包含任何的业务逻辑，break 和 continue 相当于控制流图的边，而不是节点，在控制流图的语句标注中，不将它们单独标注。另外，在包含 continue 语句的循环中，一般要求循环变量的控制语句在 continue 语句的前面，否则遇到 continue 语句，后面的循环变量将无法变更，循环将变成死循环。

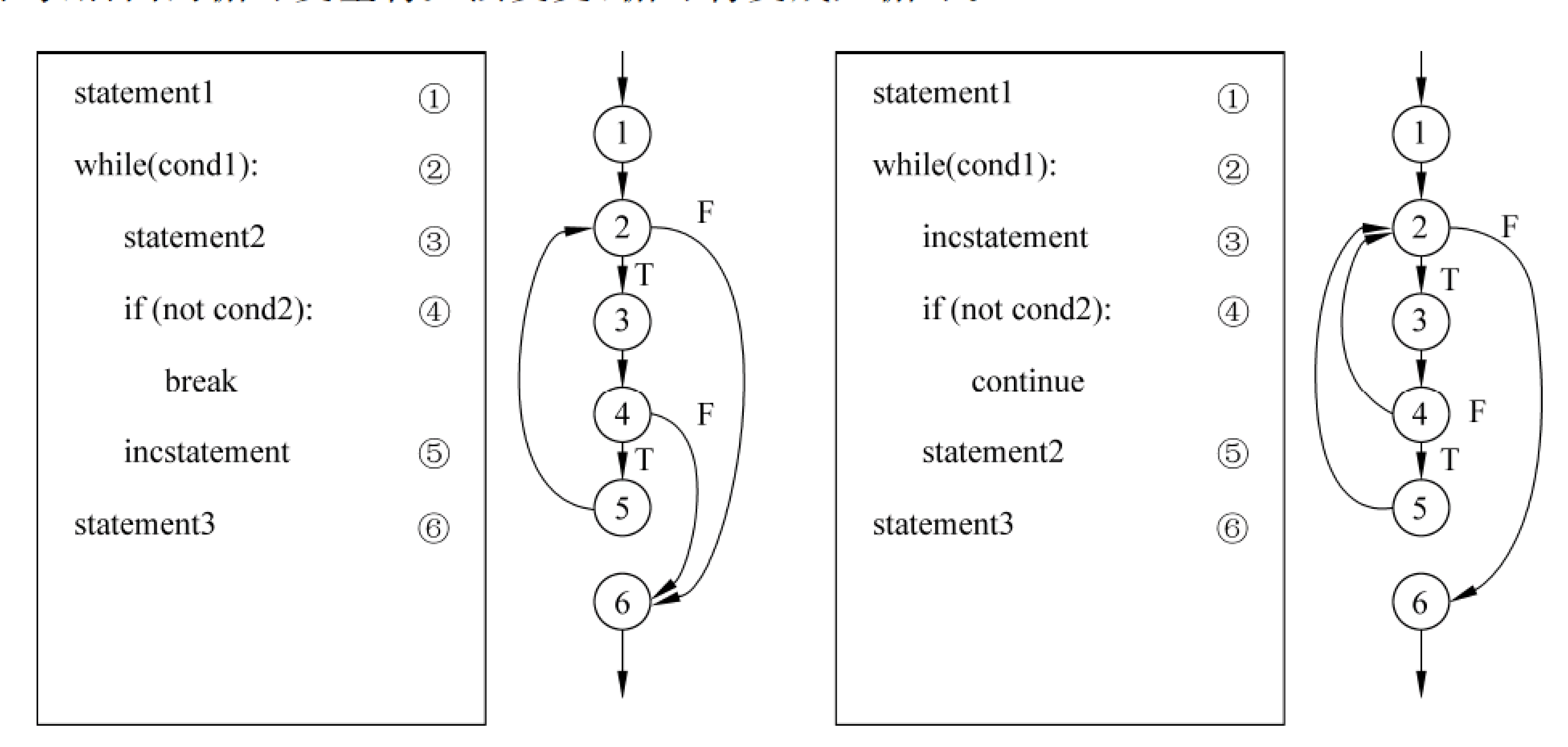


图 3-38 循环中 break 和 continue 构成的控制流图

在带有 break 语句的程序中，可能存在如下路径。

路径 1：1—2—6

路径 2：1—2—3—4—5—2—6

路径 3：1—2—3—4—6

路径 4：1—2—3—4—5—2—3—4—5—6

路径 5：1—2—3—4—2—3—4—6

……

在带有 continue 语句的程序中，可能存在如下路径。

路径 1：1—2—6

路径 2：1—2—3—4—5—2—6

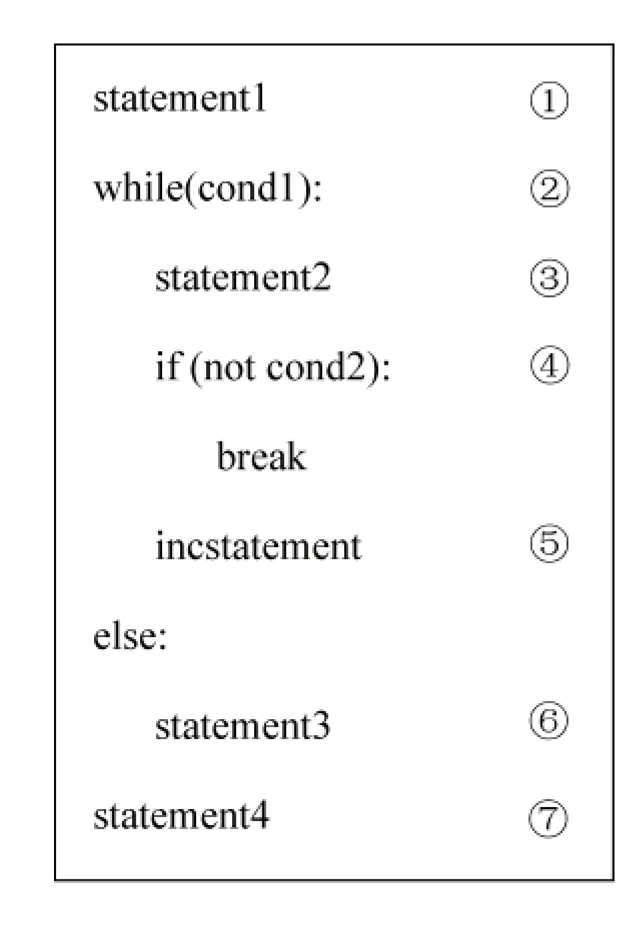
路径 3：1—2—3—4—2—6

路径 4：1—2—3—4—5—2—3—4—5—2—6

……

无论是带有 break 还是 continue 语句，在循环内部存在的分支将引起路径的指数级增长。

在 Python 中，无论是 for 循环还是 while 循环都可以带有 else 子句。else 中的语句会在循环正常执行完(即不是通过 break 跳出而中断的)的情况下执行。显然，带有 else 子句的循环体内部，一般都带有 break 语句，否则 else 中的语句体变成了必然要执行的语句体，失去了 else 语句的含义。图 3-39 给出了一个带有 else 的循环结构的控制流图，当循环体正常结束时，流程将转移到节点 6，执行 statement3 语句体，然后执行 statement4 语句体。当在循环体内部执行到 break 语句时，循环将终止，流程将跳转到节点 7，直接执行语句体 statement4。



```
statement1              ①
while(cond1):           ②
    statement2          ③
    if (not cond2):     ④
        break
    incstatement        ⑤
else:
    statement3          ⑥
statement4              ⑦
```

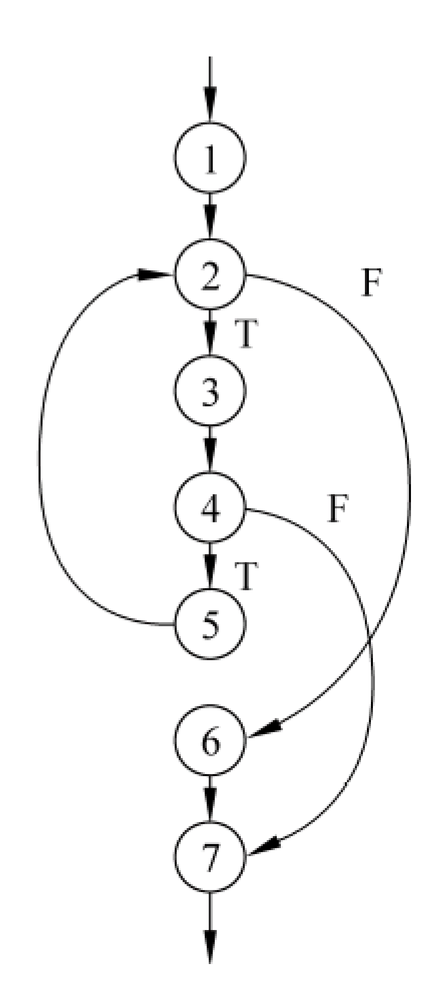


图 3-39　带 else 的循环结构的控制流图

程序 3-38 给出了一个带有 else 循环结构的例子，该例子在 100～119 之间的每一个数字判断其奇偶性，如果是奇数给出提示信息，如果是偶数，则给出其一个因式分解表达式，该表达式中，第一个因子的值最小。

程序 3-38 带有 else 的循环例子

```
#
def evenOdd():
    for num in range(100,120):
        for i in range(2,num):
            if num% i==0:
                j=num/i
                print '% d=% d * % d' % (num,i,j)
                break
        else:
            print num,' is a prime number'

evenOdd()
```

该程序由内外两个循环构成，外循环控制每一个被判断的数，内循环判断其奇偶性。执行结果如下。

```
00=2 * 50
101  is a prime number
102=2 * 51
103  is a prime number
104=2 * 52
105=3 * 35
106=2 * 53
107  is a prime number
108=2 * 54
109  is a prime number
110=2 * 55
111=3 * 37
112=2 * 56
113  is a prime number
114=2 * 57
115=5 * 23
116=2 * 58
117=3 * 39
118=2 * 59
119=7 * 17
```

3.9.2 独立路径和圈复杂度

3.9.1 节讨论了几种基本结构对应的控制流图，分析了不同基本控制流图的路径情况，在一个实际的应用程序中，实现路径的全覆盖是不现实的，折中采用独立路径（基本路径）覆盖来替代全路径覆盖。

一条路径是独立路径，那么其应满足：

(1) 是一条从入口节点到出口节点的路径；

(2) 该路径至少包含一条其他基本路径没有包含的边。

从上述定义可以看出，独立路径是相对已经选择的路径的集合而言的。对于图 3-38 中的路径集合：

路径 1：1—2—6

路径 2：1—2—3—4—2—6

路径 3：1—2—3—4—5—2—6

路径 4：1—2—3—4—5—2—3—4—5—2—6

相对于已经选择的路径 1 而言，路径 2 是独立路径，因为其包含新的边 2—3、3—4、4—2，这些都路径 1 所不包含的。若已经选定路径 1 和路径 2，则路径 3 也是独立路径，因为其包含新的节点 5，边 4—5、5—2 都是新增加的边。若选定了路径 1、路径 2、路径 3，则路径 4 不是独立路径，因为路径 4 上所有的边都在选定的路径之中。显然对于独立路径多少是和路径的选择顺序有关联的。若先选择路径 3，那么路径 1 和路径 2 都不再是独立路径。

一个控制流图中，基本路径的最大数量可以用圈复杂度来表示。圈复杂度由 Thomas J. McCabe 提出，用于度量程序的复杂性，可以通过控制流图计算出程序的圈复杂度。圈复杂度可以通过以下几个不同的公式进行计算。假设一个控制流程图 G，其圈复杂度用 $V(G)$ 表示，而 E 表示控制流图的边的数量，N 表示控制流图中节点的数量，P 为控制流图中连通图的数量。根据前面的分析，控制流图是一个连通图，所以 P 的值始终等于 1。

(1) 圈复杂度 $V(G)$ 在数值上等于控制流图有效边（不含入口边和出口边）的数量 E 减去节点的数量 N，加上 2 倍的连通图个数 P，即：

$$V(G) = E - N + 2P$$

在图 3-35 表示的顺序结构中，除了入口边和出口边以外，并无其他边，所以 $E=0$。节点数量 $N=1$，$P=1$。其复杂度为：

$$V(G) = 0 - 1 + 2 \times 1 = 1$$

在顺序结构中，无论包含多少条语句，其复杂度始终为 1。

在图 3-36 表示的单纯的 if 或者 if-else 结构中，$E=4$，$N=4$，$P=1$，其复杂度为：

$$V(G) = 4 - 4 + 2 \times 1 = 2$$

在图 3-37 表示的简单循环结构中，$E=5$，$N=5$，$P=1$，其复杂度为：

$$V(G) = 5 - 5 + 2 \times 1 = 2$$

在图 3-38 给出的具有 break 或者 continue 语句的循环结构中，$E=7$，$N=6$，$P=1$，其复杂度为：

$$V(G) = 7 - 6 + 2 \times 1 = 3$$

在图 3-39 表示的具有 else 的循环结构中，$E=8$，$N=7$，$P=1$，其复杂度为：

$$V(G) = 8 - 7 + 2 \times 1 = 3$$

实际上,从计算结果上,具有 break 或者 continue 语句的循环结构和具有 else 的循环结构在复杂度上是相同的。实际上,else 语句的不同,是由循环体内部的 if 判断中的 break 所引起的。

(2) 圈复杂度等于控制流图将平面划分区域的数量:

$$V(G) = R, \quad R \text{为区域数量}$$

所谓区域,是由控制流图的边所构成的封闭的一个区域,控制流图的外部也认为是一个区域。在顺序结构中,控制流图内部并没有构成任何的区域,仅存在控制外部的一个区域,所以其复杂度 $V(G)=R=1$。图 3-40 给出了基本循环、包含 continue 语句的循环构成的区域示意图。

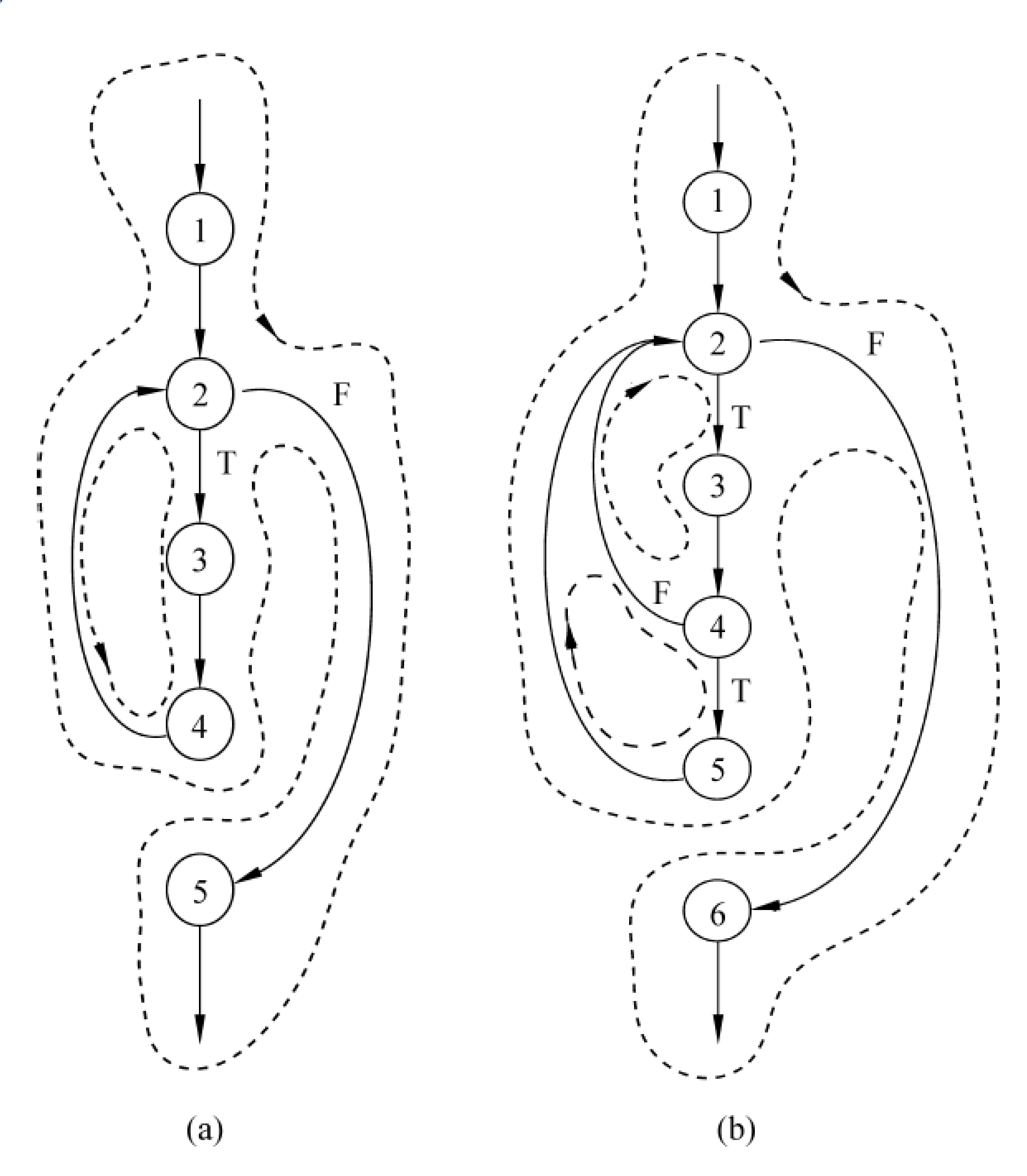


图 3-40　基本循环和包含 continue 语句循环的区域

基本循环结构图 3-40(a)中,构成两个区域,其中第一个由 2—3—4—2 节点包围构成,另一个是在控制流图的外围,故左边的基本循环的圈复杂度为 $V(G)=R=2$。

在右边的带有 continue 语句的循环结构图 3-40(b)中,存在三个区域,一个是 2—3—4—2 节点所包围,第二个是由 2—3—4—5—2 所包围,第三个是控制流图的外围。故右边的基本循环的圈复杂度为 $V(G)=R=3$。

图 3-41 给出了具有 break 语句以及 else 语句的循环结构形成的区域,图 3-41(a)为 break 语句的区域,其三个区域和具有 continue 语句的结构类似,比较清楚。图 3-39 中给出的控制流程图存在交叉线条的情况。连接节点 4 和节点 7 之间的连线、连接节点 2 和节点 6 的连线,这两条连线似乎交叉。但实际上,可以通过重新绘制连接节点 4 和节点 7 之间的连线,避免交叉,如图 3-41(b)所示。

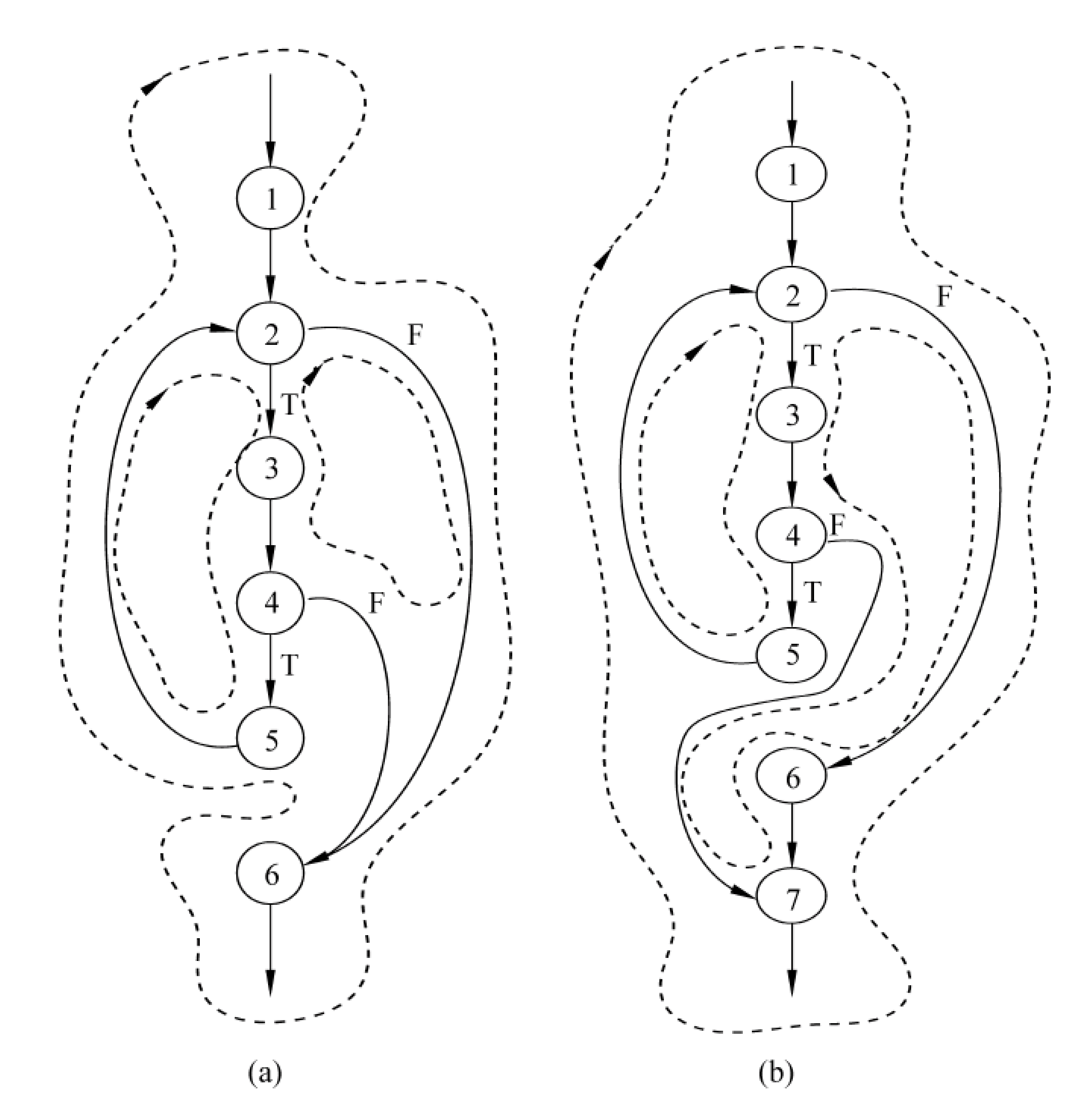


图 3-41 具有 break 和 else 结构的循环区域

(3) 圈复杂度等于控制流图中的判定节点的数量加 1:

$$V(G) = C + 1$$

其中:

$V(G)$:控制流图的圈复杂度。

C:判定节点数量,这里的判定节点假定为简单判定,就是不包含逻辑运算符。

显然在如图 3-35 所示的顺序结构中,不存在判定节点,其圈复杂度 $V(G)=1$,和前面的计算结果一致。

图 3-36 的分支结果,仅包含一个判定节点 2,其圈复杂度 $V(G)=2$,和前面的计算结果一致。图 3-37 的简单循环中,仅包含一个判定节点 2,其圈复杂度 $V(G)=2$,和前面的计算结果一致。图 3-38 和图 3-39 中间都存在两个判定节点,其圈复杂度 $V(G)=2$,和前面的计算结果一致。

若一个判定节点存在多于两个的输出分支,此时必须进行分支的拆分,使其等效于控制流图的每一个判定节点都仅包含两个输出分支。图 3-42 中,根据前面方法 1,可以计算得到:

$$V(G) = E - N + 2C = 12 - 9 + 2 = 5$$

根据区域容易看出,在节点 1 和节点 6 之间存在三个区域,在节点 6 和节点 9 之间存在一个区域,外部一个区域,显然:

$$V(G) = 3 + 1 + 1 = 5$$

显然前面两种方法计算得出的结果是一样的。若不进行判定的转换，则仅有节点 1 和节点 6 为判定结果，但是判定节点 1 处在 4 路输出，将该判定节点转化为等价的 2 路分支输出，如图 3-42(b)所示，则判定节点包含节点 1、节点 11、节点 111、节点 6。因此得到的全复杂度为：

$$V(G)=P+1=5$$

另外一种情况，即使输出的路径只有两个输出，但是包含组合条件而形成的判定，也需要将判定改写成为等效的简单判定。

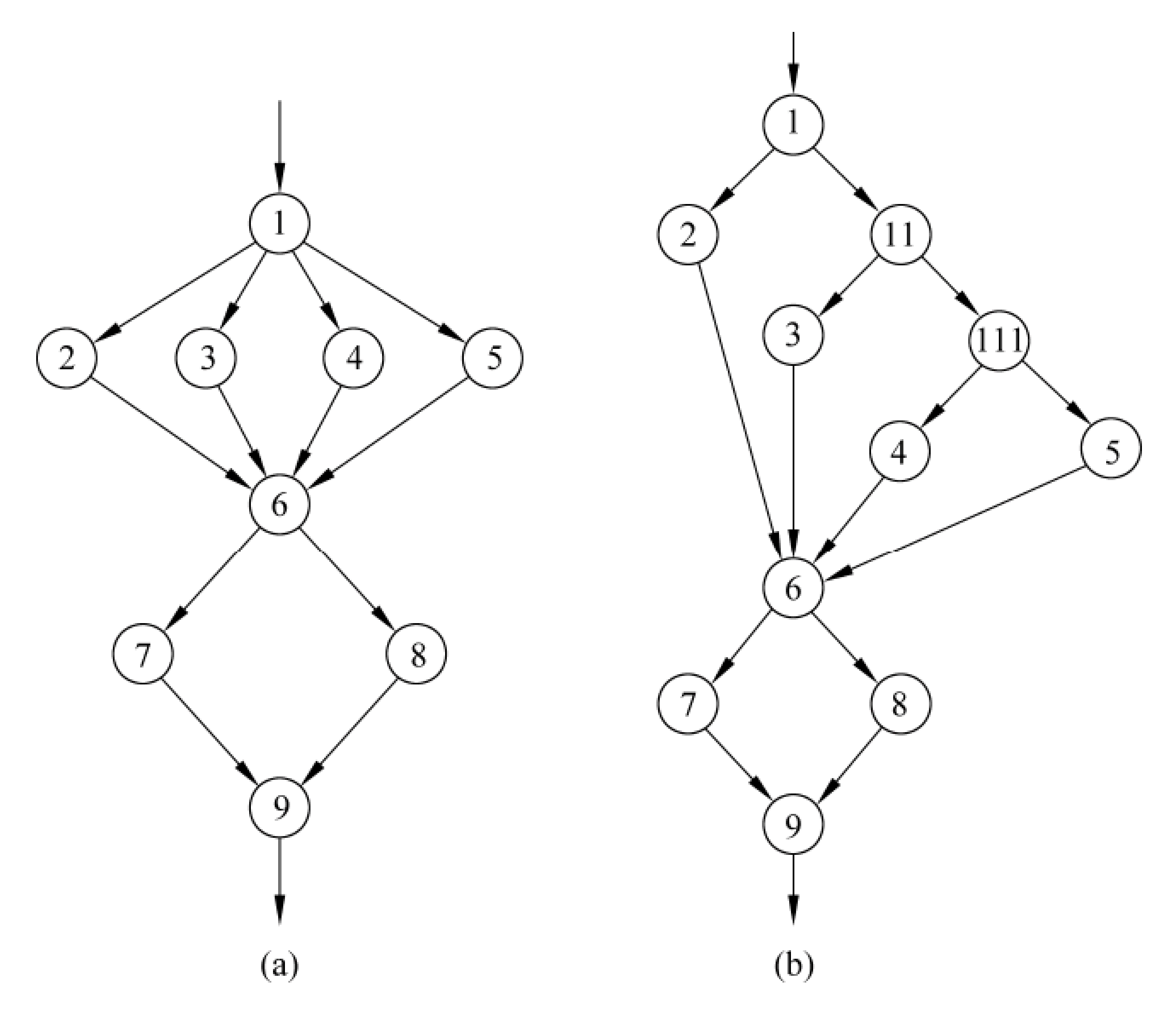


图 3-42 具有多输出分支的判定转换求圈复杂度

例如：

```
if  (A>=MIN and A<=MAX):
    dosomething()
else:
    dosthelse()
```

这个判定中由 and 运算符连接了两个条件，其实等效于：

```
if(A<MIN):
    dosthelse()
elseif(A>MAX):
    dosthelse()
else:
    dosomething()
```

上面讨论了三种圈复杂度的基本计算方法，一个软件随着圈复杂度的不断递增，程序的可读性会随之递减，而维护成本也会随之增加，出错的概率会随之增长。从软件质量的

角度看,要尽可能地降低软件的圈复杂度。从测试的角度看,一个圈复杂度高的软件或者模块,应设计更多的测试用例。

3.9.3 基本路径覆盖

执行基本路径测试通常需要包括以下4个步骤。

(1) 依据程序的基本结构,分析其路径个数,画出程序的控制流图。

(2) 根据控制流图,计算出程序的圈复杂度。

(3) 根据控制流图,构造程序的独立路径集合,其上限为圈复杂度。

(4) 根据(3)中的独立路径,设计测试用例的输入数据和预期输出。

有个数据列表,求出其数值在min和max之间的数据平均值。其Python代码如程序3-39所示。试用基本路径法设计测试用例。

程序3-39 求列表平均值程序

```
#average.py
def average(values,min,max):
    i=0;
    validNum=0;                                   ①
    sum=0;
    while(i<len(values)  and \                    ②
          values[i]!=-999):                       ③
        if(values[i]>=min and \                   ④
           values[i]<=max):                       ⑤
            validNum+=1
            sum+=values[i];                       ⑥
        i=i+1;                                    ⑦
    if(validNum>0):                               ⑧
        mean=sum/validNum                         ⑨
    else:
        mean=-999                                 ⑩
return mean                                       ⑪
```

1. 画出控制流程图

在这个程序中,其主要功能由一个while循环和两个if判定构成。其中,while的循环条件和第一个if语句中的判定都是复合判定,需要将这两个条件进行分拆。在代码中,通过换行符\将两个条件写在两行上,分别标注为两个行。else子句始终和if相对应,只表示分支方向,无须单独标注。多个没有分支的初始化语句,标注为1。依据上述编码,画出控制流图,如图3-43所示。

2. 计算圈复杂度

根据控制流图,很容易计算出其圈复杂度。在图3-43中,形成区域都已经用虚线表

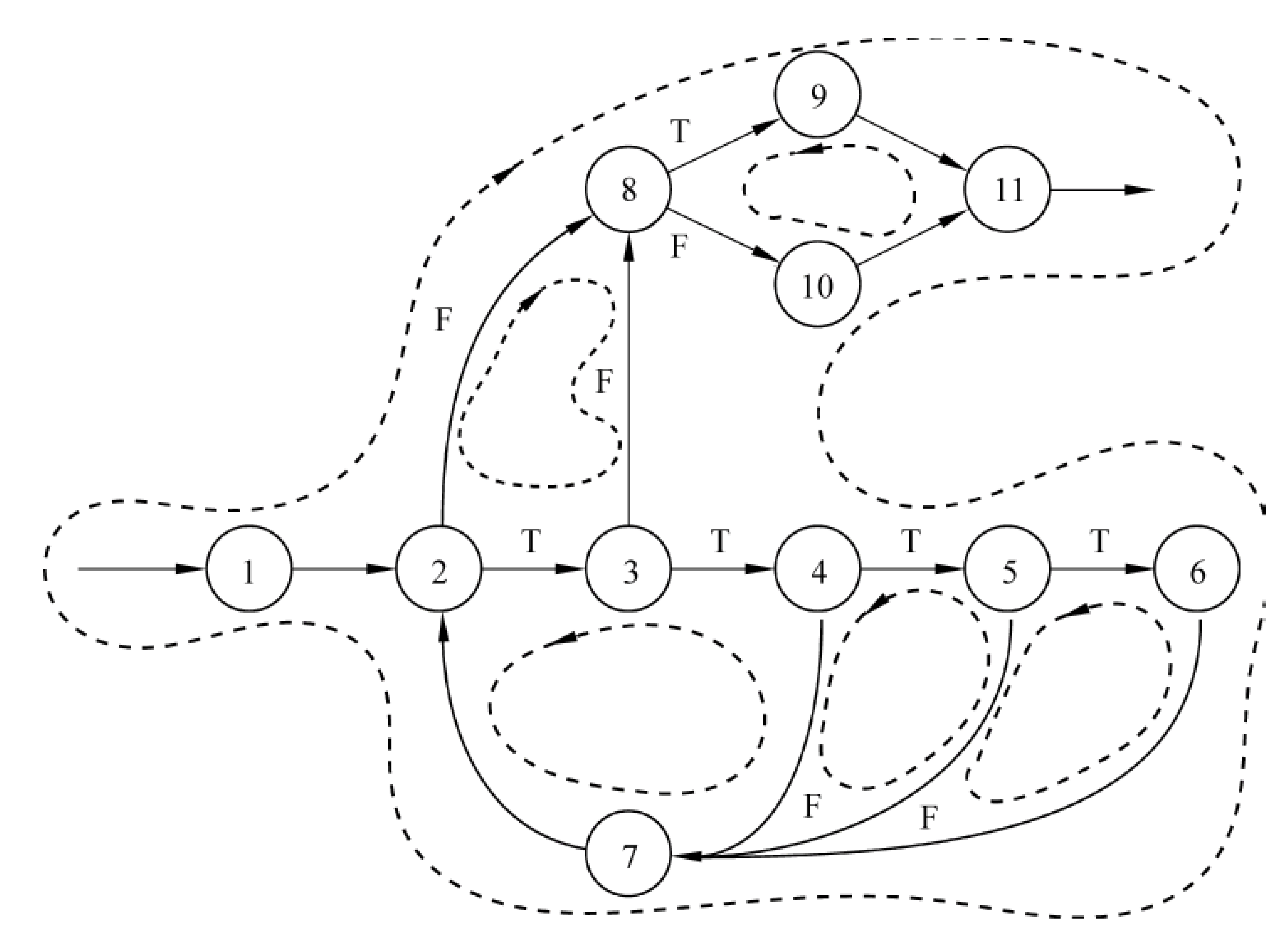


图 3-43　平均值问题的控制流图

示出来了。显然：

$$V(G) = R = 6$$

用判定节点数加以验证，在图 3-43 中判定节点为节点 2、节点 3、节点 4、节点 5、节点 8 共 5 个，所以：

$$V(G) = C + 1 = 5 + 1 = 6$$

均值问题的圈复杂度为 6。

3. 查找基本路径序列

查找基本路径的方法，一般从入口边开始一直到出口边。在本题中，查找以节点 1 为开始、节点 11 为结束的路径。一般而言，从最短的路径开始，将其加入到基本路径序列中。因为不同次序会影响基本路径序列的构成，所以不用集合表示多个基本路径构成测试集合。

在控制流图中，只是表示可能的流程走向，有些路径是实际上不可行的。

例如，路径：

1—2—8—9—11

当 values 列表为空时，i<len(values)不成立，节点 2 的结果为 F。此时 validNum 的值始终为 0，节点 8 中的 validNum>0 始终为 F，流程永远不会达到节点 9。所以路径 1—2—3—8—9—11 这种路径是实际上不可行的，但是简单地从控制流图，无法发现这种不可行的路径。

选择第一条最短的独立路径为：

1—2—8—10—11

在第一条路径中，考虑节点 3，新增加 2—3 以及 3—8 两条边，形成第二条独立路径：

1—2—3—8—10—11

在第二条独立路径的基础上，考虑节点 4，新增加边 3—4、4—7、7—2，形成第三条独立路径：

1—2—3—4—7—2—8—10—11

同时，考虑节点 5，形成独立路径：

1—2—3—4—5—7—2—8—10—11

考虑节点 6 时，显然 validNum>0 是始终成立的，节点 8 的结果永远为 True，所以最短的独立路径为：

1—2—3—4—5—6—7—2—8—9—11

该独立路径仅考虑只有一个元素的情况。但在一般情况下，需要考虑多个数据元素。在这个独立路径中，[2—3—4—5—6—7]是可以重复的。

另外，在列表中，-999 作为计数结束的另一个重要标准，可增加一个以 -999 作为结束标志的测试用例。

自此，控制流图中的所有边都已经被覆盖。综上分析，形成的测试路径如下。

路径 1：1—2—8—10—11

路径 2：1—2—3—8—10—11

路径 3：1—2—3—4—7—2—8—10—11

路径 4：1—2—3—4—5—7—2—8—10—11

路径 5：1—[2—3—4—5—6—7]—2—8—9—11

路径 6：1—[2—3—4—5—6—7]—2—3—8—9—11

4. 构造测试用例

确定路径 1 的测试用例，由于在这个路径上 i 初始值为 0，关键所在是节点 2，输入的列表 values 为空列表，此时 min 和 max 的值不影响输出结果，期望的输出结果为 -999。

路径 2 是在节点 2 结果为 True 的情况下，考虑 values[i]! = -999 不成立，而 validNum>0 不成立的情况。综合三点判定的情况，要求列表有且仅有一个元素，该元素的值必须为 -999，此时 min 和 max 的值不影响输出结果。期望的输出为 -999。

路径 3，列表仅允许有一个元素，且该元素的值小于最小值 min，使得节点 8 的结果为 False，期望的输出为 -999。

路径 4 和路径 3 类似，列表仅允许有一个元素，且该元素的值大于最大值 max，使得节点 8 的结果为 False，期望的输出为 -999。

路径 5 表示非空列表，且列表至少有一个值在 min 和 max 之间，且不包含 -999。路径 6 表示列表有多个元素，至少有一个值在 min 和 max 之间，且包含 -999。

假定 min 的值为 0，max 的值为 100，在 min 和 max 之间的值为 70。最终形成的测试如程序 3-40 所示。

程序 3-40 基本路径测试程序

```
mport  pytest
from app.average import average
```

```
@pytest.mark.parametrize("vlist,min,max,result", [
    ([],0,100,-999),
    ([-999],0,100,-999),
    ([-5],0,100,-999),
    ([102],0,100,-999),
    ([75,85,80],0,100,80),
    ([70,90,80,-999,88],0,100,80),
])
def test_average(vlist,min,max,result):
    assert average(vlist,min,max)==result;
```

从该例子可以看出，路径测试和前面的测试不同，其关注整体的逻辑关系，特别是前后不同判定或者条件之间的相互影响。

第4章　组合测试

除了单个因素导致的软件缺陷外，很多缺陷往往是由多个因素交互而导致的。在实际应用中，因素之间交互而导致的缺陷比单因素的缺陷更难发现。组合测试是一种测试用例生成方法。它将被测应用抽象为一个受到多个因素影响的系统，其中每个因素的取值是离散且有限的。两因素组合测试生成一组测试用例集，可以覆盖任意两个因素的所有取值组合。多因素组合测试可以生成测试用例集，以覆盖任意 N 个因素的所有取值组合。本章重点讨论利用正交表产生测试、成对组合测试、可变强度和具有约束的组合测试等方法和技巧。

4.1　多参数的故障模型

软件系统行为受到很多因素的影响，例如输入参数、环境配置和状态变量。利用等价类划分或者边界值分析的方法可确定不同因素可能的取值。

例如，Ubuntu 12.04 服务器版系统的 more 命令，其命令参数列表如图 4-1 所示。共有 11 个可选项，寻找能够满足测试需要的关系。在这些选项中，8 个选项是开关型的选项，比较容易确定参数的输入范围，其他几个选项先采用其他模式确定几个类型的输入类别。

```
clz@ubuntu:~$  more
Usage: more [options] file…
Options:
-d          display help instead of ring bell
-f          count logical, rather than screen lines
-l          suppress pause after form feed
-p          suppress scroll, clean screen and display text
-c          suppress scroll, display text and clean line ends
-u          suppress underlining
-s          squeeze multiple blank lines into one
-NUM        specify the number of lines per screenful
+NUM        display file beginning from line number NUM
+/STRING    display file beginning from search string match
-V          output version information and exit
```

图 4-1　Ubuntu 中的 more 命令参数

从简单的参数含义来看，不同的参数之间相互作用是不一样的，例如-V 参数，仅显示版本信息，和显示效果没有直接关系，而其他对于显示的影响程度是不一样的，如表 4-1 所示。

表 4-1 more 命令的参数及其含义

参数	含义	影响显示行
-d	提示“Press space to continue,'q' to quit”,禁用响铃功能	否
-f	计算行数时,以实际的行数计算,而非自动换行过后的行数(有些单行字数太长的会被扩展为两行或两行以上)	是
-l	忽略 Ctrl+l(换页)字符	是
-p	通过清除窗口而不是滚屏来对文件进行换页,与-c 选项相似	否
-c	从顶部清屏,然后显示	否
-u	把文件内容中的下划线去掉	否
-s	把连续的多个空行显示为一行	是
−NUM	定义屏幕大小为 NUM 行	是
+NUM	从第 NUM 行开始显示	是
+/STRING	在每个档案显示前搜寻该字串(STRING),然后从该字串前两行之后开始显示	是
-V	输出版本信息	否

在实体的操作环境中也存在类似的问题。例如,在美国航天飞机驾驶舱中存在大量的开关和操作按钮,如图 4-2 所示。这些开关可能会存在相互作用,在测试时必须将所有存在相互作用的开关或者按钮可能的组合都覆盖到。若系统具有 34 个开关,每个开关具有两种状态。根据全组合,那么需要的测试用例数量为:

$$2^{34} = 1.7 \times 10^{10}$$

如此庞大的测试用例数量,其测试成本是非常高昂的。



图 4-2 美国航天飞机驾驶舱内景

WPS 的字体设置情况类似。在 9.10 版的 WPS 文字中的字体设置中的效果设置,共有 11 个设置参数,包括删除线、阴影等,每个参数有选择和未选择两种状态,如图 4-3 所

示。这些设置之间存在较多的交互关系，也将产生巨大的测试用例集合。



图 4-3　WPS 的字体设置中的效果设置

美国国家标准和技术研究院(NIST)研究了组合测试浏览器、服务器软件、NASA 分布式数据库、医疗设备 4 类系统的错误检测率，如图 4-4 所示。以 NASA 分布式数据系统为例，67%的错误是由单参数测试发现的，二路参数组合测试发现 93%的缺陷，三路参数组合测试发现 98%的缺陷。在其他系统中，组合测试缺陷发现曲线都非常类似，在 6 参数组合下，几乎可以达到 100%。如何在显著不降低质量的前提下降低测试数量是问题的出发点。美国国家标准和技术研究院(NIST)研究发现多种组合的测试效果随着组合强度的增加快速增长，大量的测试结果表明，在一般情况下，四路到六路组合已经满足大部分测试需求。

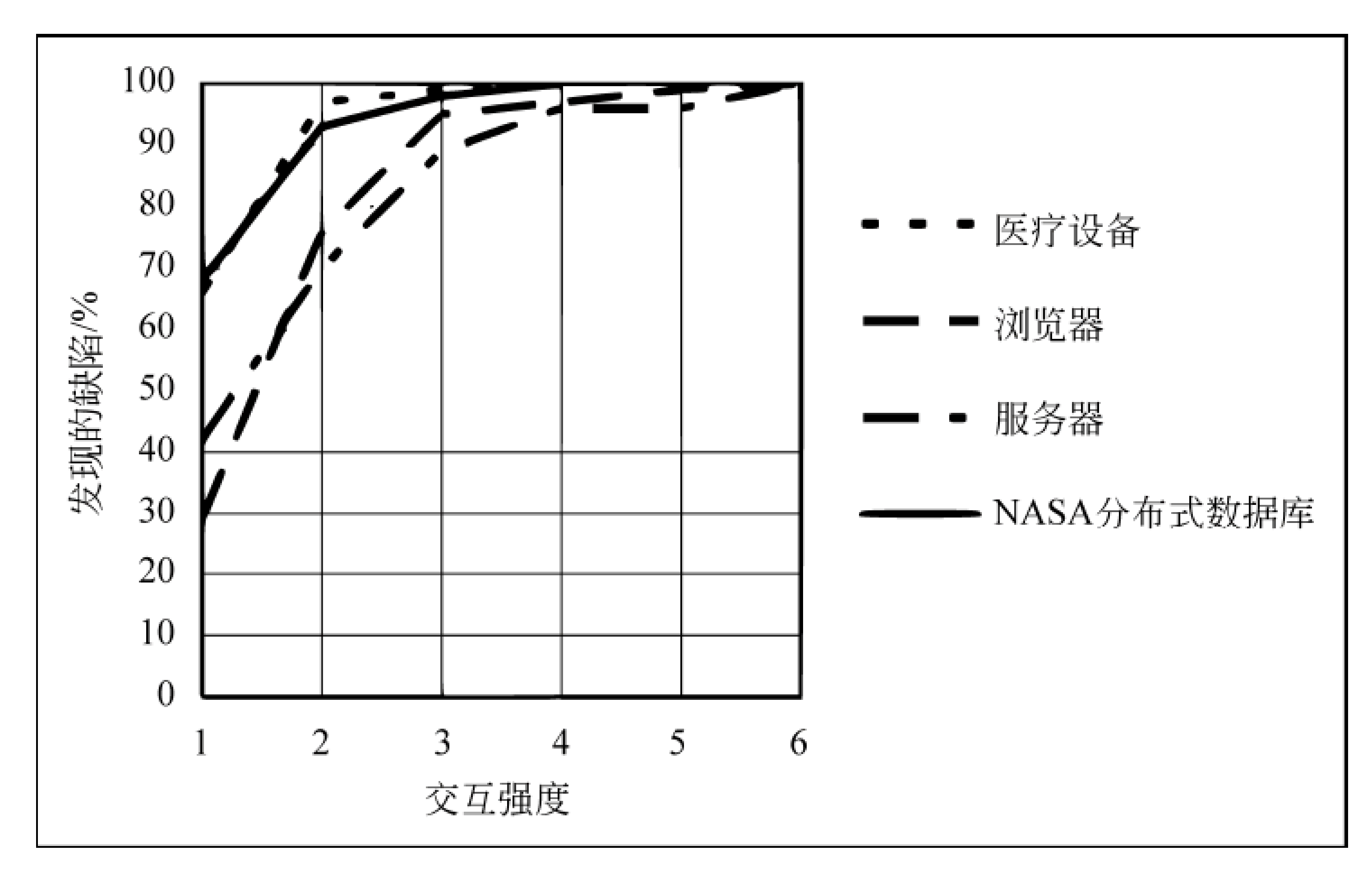


图 4-4　发现缺陷和组合强度的关系

4.2　利用正交表实现测试

4.2.1　拉丁方阵

古希腊是一个多民族的国家，国王在检阅臣民时要求每个方队中每行有一个民族代表，每列也要有一个民族的代表。数学家在设计方阵时，以每一个拉丁字母表示一个民

族，所以设计的方阵称为拉丁方阵。

拉丁方阵的定义：用 n 个不同的拉丁字母(数字)排成一个 n 阶方阵，如果每行的 n 个字母均不相同，每列的 n 个字母(数字)均不相同，则称这种方阵为 $n\times n$ 拉丁方阵或 n 阶拉丁方阵。每个字母在任一行、任一列中只出现一次。即 n 阶方阵的每行、每列构成 $\{1,2,3,\cdots,n\}$ 的两个全排列。

例 4-1 4 个数字 1、2、3、4 构成的拉丁方阵：

$$A=\begin{matrix}1&2&3&4\\2&3&4&1\\3&\underline{4}&1&2\\4&1&2&3\end{matrix}$$

一个 n 阶拉丁方阵包含 n 个元素值，每一个元素同时包含一个纵坐标和一个横坐标，元素值、横坐标、纵坐标三个取值范围均为 $1\sim n$，将三者合成构成一个三元组(行号，列号，元素值)，这个三元组恰好能够表示三个参数的组合。在例 4-1 中第三行，带有下划线的 4，可以用三元组(3,2,4)表示。因此一个 n 阶拉丁方阵可以表示为三个参数，每个参数具有 n 个候选值的测试用例。整个拉丁方阵表示的测试用例，可以用表 4-2 表示。

表 4-2 4 阶拉丁方阵表示的测试用例

用例编号	参数 1	参数 2	参数 3
1	1	1	1
2	1	2	2
3	1	3	3
4	1	4	4
5	2	1	2
6	2	2	3
7	2	3	4
8	2	4	1
9	3	1	3
10	3	2	4
11	3	3	1
12	3	4	2
13	4	1	4
14	4	2	1
15	4	3	2
16	4	4	3

若被测函数的输入参数大于三个，则需要用正交拉丁方阵来表示。

正交拉丁方阵：设 $A=(a_{ij})$，$B=(b_{ij})$ 是两个 n 阶拉丁方阵，如果由 A 和 B 对应元素构成所有的有序对(a_{ij}, b_{ij})两两不同，则称 A 和 B 为一对正交的 n 阶拉丁方阵。

例 4-2 下面的拉丁方阵 A 和 B 分别为三阶正交拉丁方阵。

$$A = \begin{matrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{matrix} \quad B = \begin{matrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{matrix}$$

因为由 A 和 B 构成的组合方阵：

$$C = \begin{matrix} (1,1) & (2,2) & (3,3) \\ (2,3) & (3,1) & (1,2) \\ (3,2) & \underline{(1,3)} & (2,1) \end{matrix}$$

C 中每一个元素都是由方阵 A 和 B 对应的元素构成有序对，并且互不相同。可以验证，由三个元素构成的正交拉丁方阵最多只有两个。

在一个由 t 个拉丁方阵组合而成的方阵中，每一个元素为由单个方阵元素构成的有序组合（含有 t 个元素），每一个元素也存在一个纵坐标和横坐标，它们可以进一步组合成 $t+2$ 个元素的元组（行号、列号、有序组合）。在例 4-2 中构成的组合方阵 C 中，在第三行第 2 列带下划线的元组(1,3)，其行号为 3，列号为 2，连同其坐标可以构成组合(3,2,1,3)。该组合可以表示有 4 个参数的被测函数的测试用例。

由组合方阵 C 表示的完整测试用例，可以测试具有 4 个参数的函数，每一个参数具有三个候选值的函数或者配置，如表 4-3 所示。

表 4-3 测试具有 4 个参数的拉丁方阵

编号	参数 1	参数 2	参数 3	参数 4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

若有 t 个 n 阶拉丁方阵，若它们两两正交，则它们组成一个 n 阶(t 个)正交拉丁方阵组。若以 $\mathcal{N}(n)$表示 n 阶正交拉丁方阵组包含最多的拉丁方阵个数，已经证明 $\mathcal{N}(n) \leqslant n-1$。对于 n 阶拉丁方阵，若存在 $n-1$ 个正交拉丁方阵，则称其为 n 阶正交拉丁方阵完备组。

对于任何的正整数 n，都存在 n 阶拉丁方阵，但不一定存在 n 阶的正交拉丁方阵。

例如，二阶拉丁方阵只有两个：

$$A=\begin{matrix}1 & 2\\2 & 1\end{matrix}\quad B=\begin{matrix}2 & 1\\1 & 2\end{matrix}$$

显然 A 和 B 不正交。

从上面的原理分析可以知道，应用正交拉丁方阵设计测试用例存在以下几个方面的限制。

(1) 部分正交拉丁方阵不存在。若没有对应的拉丁方阵，可以扩大拉丁方阵阶数。

(2) 一般要求参数的候选值的个数相同。如果候选值的个数不相同，只能选取候选个数最多的参数值个数作为统一的个数。

(3) 参数的个数远远大于参数值个数。正交拉丁方阵的大小受参数值的个数限制，也可能不存在对应的正交拉丁方阵。

4.2.2 正交表

假设需要测试的函数由 A,B,C,D 4 个参数的取值决定，每个参数都有 10 个不同的取值，若把每个取值都测试一遍，需要执行 10×10×10×10=10 000 次测试。为了减少测试用例的数量，必须选定部分有代表性的测试用例。正交表就是其中一种典型的技术手段。

正交试验设计是研究多因素多水平的一种设计方法，它根据正交性从全面试验中挑选出部分有代表性的点进行试验，这些有代表性的点具备“均匀分散，齐整可比”的特点，是一种高效率、快速、经济的实验设计方法。

日本著名的统计学家田口玄一将正交试验选择的水平组合列成表格，称为正交表。用正交表设计出来的试验方案具有如下两个重要的特征：均衡搭配——正交性可以用较少的试验次数替代全部可能试验组合中好的、中等的、不好的搭配组合，使选出的较少的搭配组合具有均衡的代表性。通过试验数据的适当组合，可发现各组试验数据以及各因素影响之间的某种可比性。

设有两组元素 $a_1,a_2,\cdots,a_n$ 与 $b_1,b_2,b_3,\cdots,b_k$，它们可构成如下的元素对。

$$\begin{matrix}(a_1,b_1) & (a_1,b_2) & \cdots & (a_1,b_k)\\(a_2,b_1) & (a_2,b_2) & \cdots & (a_2,b_k)\\\vdots & \vdots & \vdots & \vdots\\(a_n,b_1) & (a_n,b_2) & \cdots & (a_n,b_k)\end{matrix}$$

称这些元素对是由元素 $a_1,a_2,\cdots,a_n$ 与 $b_1,b_2,b_3,\cdots,b_k$ 构成的完全有序元素对，简称元素对。

例 4-3 由数字(1,2,3,4)和(1,2,3)构成的完全有序数字对：

(1,1) (1,2) (1,3)

(2,1) (2,2) (2,3)

(3,1) (3,2) (3,3)

(4,1) (4,2) (4,3)

若在一个矩阵中的任意两列中，由两列中对应的数字是完全元素对并且每对出现的次数相等，则称这两列是均衡搭配。

假设有矩阵 **A**：

$$\mathbf{A}=\begin{bmatrix}1&1&1\\1&1&2\\1&2&1\\1&2&2\\2&1&2\\2&1&2\\2&2&2\\2&2&2\end{bmatrix}$$

在矩阵 **A** 中，第一列和第二列中对应的元素构成 8 个数字对：(1,1)，(1,1)，(1,2)，(1,2)，(2,1)，(2,1)，(2,2)，(2,2)，是由第一列中的 1 和 2 以及第二列中的 1 和 2 构成的完全数字对，每对均出现两次，因此称这两列为均衡搭配。

在介绍正交表测试之前，下面先介绍因素、水平、强度和正交表的定义。

因素：在一项试验中，凡欲考察的变量称为因素（变量）。在软件测试中，是指有多少个影响输出的参数、变量或者配置。

水平：任何单个因素能够取得的值的个数。在软件测试中，是指一个参数、变量或者配置允许取值的个数。

强度：变量间的相互关系。当强度为 2 时，只考虑变量两两之间的影响，如果强度为 3，同时考虑三个变量对结果的影响；当强度增加时，测试用例的个数会急剧增加。

次数：实验次数的多少，在测试中就是指多少个测试用例。

正交表：设 A 是一个 $n\times k$ 的矩阵（n 行 k 列），其中，j 列由元素（$1,2,\cdots,n$）构成（$j=1,2,\cdots,k$），若 A 的任意两列均衡搭配，则称 A 是一张正交表。

正交表一般表示成：

$$L_n(m_1\times m_2\times m_3\times\cdots\times m_k)$$

式中，L 为正交表的代号，是拉丁方阵 Latin Square 的首字母；k 为正交表的列数，每一列对应着一个实验因素；n 为正交表的行数，表示实验的次数；m_j 为第 j 列中元素的个数，表示第 j 个因素的水平数。

例如，正交表 $L_8(4\times2\times2\times2\times2)$ 表示第 1 列的水平数为 4，后面 4 列的水平数为 2，实验次数为 8。这个正交表可以表示一个函数的测试用例集，该函数共具有 5 个参数，第 1 个参数有 4 个不同的取值，而后 4 个参数各有两个不同的取值，共执行 8 次测试。

在正交表中，若某些列的元素格式相同，则可以写成指数形式：

$$L_n({l_1}^{f_1}\times{l_2}^{f_2}\times\cdots\times{l_i}^{f_i})$$

式中，n 表示次数，l_i 表示水平数，f_i 表示因素数个数。例如，前面的正交表可以表示成 $L_8(4\times2^4)$。

各个水平数不完全相同的正交表称为混合水平正交表。如 $L_{16}(4^4\times2^3)$，$L_{16}(4\times2^{12})$ 等都是混合水平正交表。

在不显著降低检测能力的前提下，利用正交表可以显著减少测试用例数量。例如，作一个三因素三水平的实验，按全面实验要求，须进行 $3^3=27$ 种组合的实验，且尚未考虑每一组合的重复数。若按 $L_9(3^3)$ 正交表安排实验，只需作 9 次实验，显然大大减少了工作量。因而正交实验设计在很多领域的研究中已经得到广泛应用。

例 4-4 有一个函数，其输入有三个变量，每个变量分别有两种取值，有 4 个测试用例，即 $n=8$。那么其对应的正交表可以表示为 $L_4(2^3)$，假设采用 1，2 分别表示不同的变量取值，一个满足该条件的正交表如图 4-5 所示。

1	1	1
1	2	2
2	1	2
2	2	1

图 4-5 一个三个变量的正交表示例

容易验证其满足正交性。

例 4-5 有一个函数，其输入有 5 个变量，前 4 个变量，每个变量分别有两种取值，第 5 个变量有 4 种取值。如果有 8 个测试用例，即 $n=8$，那么其对应的正交表可以表示为 $L_8(2^4\times4)$。假设采用 0，1，2，3 分别表示不同的变量取值，一个满足该条件的正交表可以表示为如图 4-6 所示。

0	0	0	0	0
0	0	1	1	2
0	1	0	1	1
0	1	1	0	3
1	0	0	1	3
1	0	1	0	1
1	1	0	0	2
1	1	1	1	0

图 4-6 一个 5 个变量的正交表示例

现以第 4 个因素和第 5 个因素为例，校验表格正交性。将下面左边的矩阵根据第一列从 0 到 1 排序，然后将左边第二列根据 0，1，2，3 顺序排列，形成右边的矩阵。可以清晰看出其正交性。

$$\begin{bmatrix} 0 & 0 \\ 1 & 2 \\ 1 & 1 \\ 0 & 3 \\ 1 & 3 \\ 0 & 1 \\ 0 & 2 \\ 1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \\ 0 & 3 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix}$$

4.2.3 正交表的性质

正交表具有如下两个性质。

性质1：任意列中各水平重复出现的次数相等。第 j 列各水平重复出现的次数：

$$t = n/m_j$$

性质1所表示的含义为，在同一张正交表中，每个因素的每个水平出现的次数是完全相同的，所以由正交表所构成的测试用例，一个变量各个输入（或者配置）值出现次数相同。由于在试验中每个因素的每个水平与其他因素的每个水平参与试验的概率是完全相同的，这就保证在各个水平中最大程度地排除了其他因素水平的干扰。因而，能最有效地进行比较，找到好的试验条件。

性质2：任意两列所构成的水平对是完全有序数字对，各个水平重复出现的次数相同（均衡搭配）。第 i 列和第 j 列所构成水平对重复的次数为：

$$s = \frac{n}{m_i \times m_j} \quad (i \neq j)$$

性质2表示，在同一张正交表中，任意两列（两个因素）的水平搭配（横向形成的数字对）是完全相同的，所以正交表所构成的测试用例中，任意两个变量各个输入（或者配置）值对出现次数相同，保证了试验条件均衡地分散在因素水平的完全组合之中，因而具有很强的代表性，容易得到好的试验条件。

为更直观地描述正交试验设计与正交表的特点，下面用一个立方体图来说明正交表测试分布的均匀性。假设有个函数的测试需要三个输入变量（配置），每个变量有三种可能的取值，全部覆盖需要做27次测试，如图4-7所示。这27次测试相当于图4-7中立方体各条线的交点，经正交试验设计后，正交试验的9个测试均匀分布在立方体的各个部

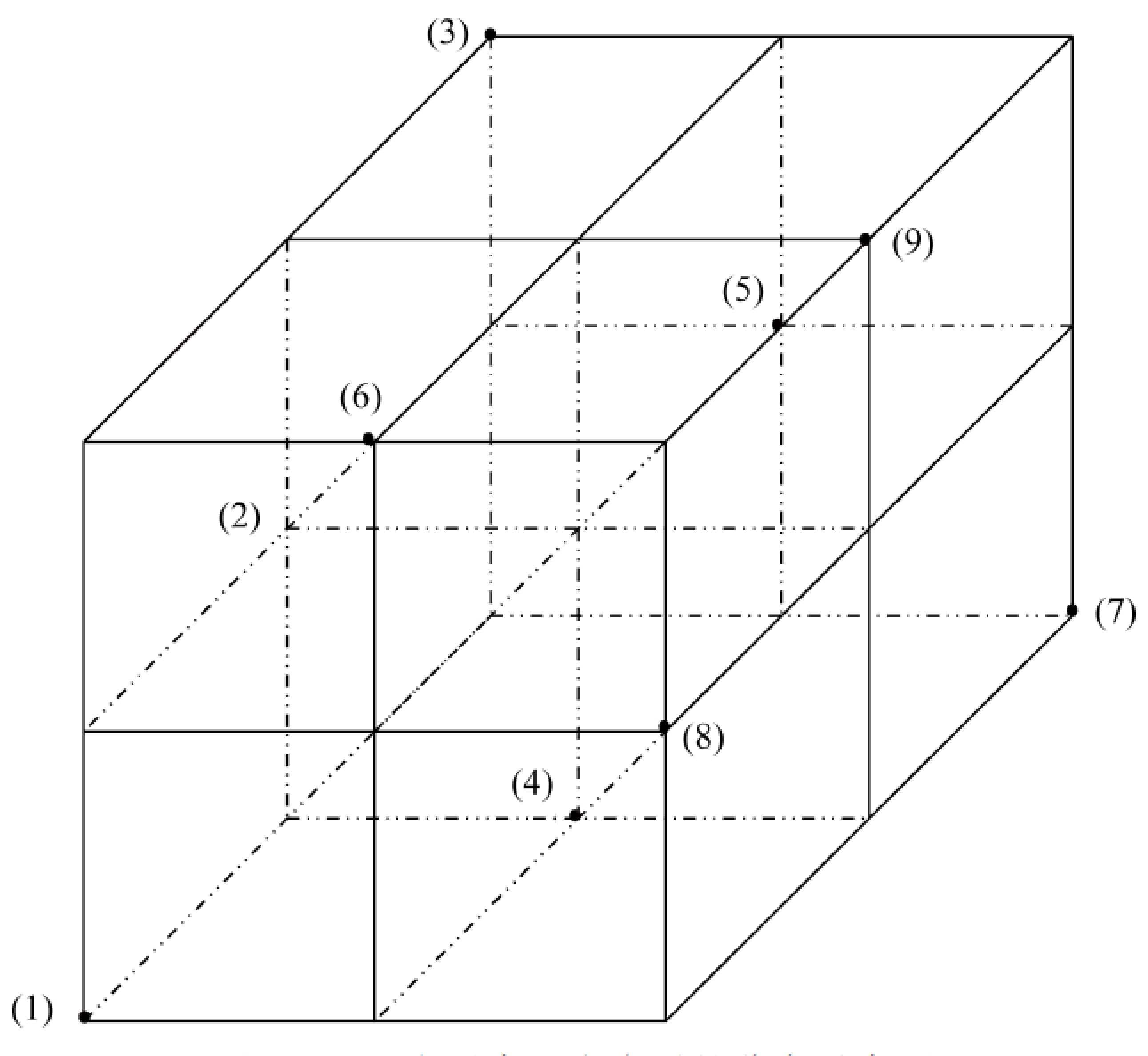


图4-7 三个因素三个水平的分布示意图

位。在上、中、下、左、中、右、前、中、后的9个面上均衡整齐地分布着三个试验点，在27条线上，每条线上分布着一个点，非常均匀。因此，用这9个点进行测试基本上反映了27个测试的情况。

根据正交表的性质，可以得到以下三种变换。

（1）列间置换：正交表中任意两列可以相互交换。

（2）行间置换：正交表中任意两行可以相互交换。

（3）水平置换：正交表中任意一列中的水平数字可以相互交换。

如果一个正交表经过上述变换以后仍为正交表，则变换以后的正交表称为原正交表的等价表。对于具有等价表的正交表，在测试领域的含义为：由正交表构成的测试用例，输入变量（参数）之间没有顺序关系。两个测试用例没有顺序关系。一个变量的参数取值没有顺序关系。

正交表优雅的特性，使得其在各个领域均有着广泛的应用，然而正交表的构造并非一件容易的事，不同的正交表有不同的构造方法，有些正交表是否存在目前也没有得到证实。在附录A中列举了部分正交表供读者参考。

4.2.4 正交表测试

在应用正交表设计测试时，一般包括以下步骤。

（1）确定每一个被测函数输入参数或者配置因素的取值个数。这里的取值个数，通过其他的测试方法，例如边界值、等价类等方法确认的取值个数，是离散的、有限个取值，并非是变量值域范围的取值个数，因为在连续的值域范围内，其本身输入个数可能是无限的。

（2）确定被测函数的输入参数个数，或者配置测试的因素个数，将其作为正交表的因素数。

（3）依据步骤(1)和步骤(2)确定的因素数和水平数，选择合适的正交表。由于并非所有因素数和水平数的组合都存在正交表，当不存在对应的正交表时，应选择能够包含其因素和水平的正交表。可能存在以下三种情况。

① 存在因素数和水平数刚好符合被测问题的正交表。

② 因素数不相同：水平数相同，但是找不到相同因素数的正交表，可以选择因素数最接近，但略大的实际因素数的正交表。

③ 水平数不相同：因素的水平数不相同，但是可以选择水平数最接近但是略大的正交表。

（4）如果所选择的正交表中某个因素有多余的水平数，可以应用这个因素的可选值，进行填充，以增加发现缺陷的机会。填充的原则可以包括：

① 随机填充，任意选择因素的一个值进行填充。

② 依据因素不同可选值，可能发现问题的几率进行填充，例如边界值。

③ 依据该因素值和其他因素值的交互方式进行填充。

④ 依据行业意见的经验进行填充。

(5) 在此基础上,可以根据测试经验增加若干个测试。

例 4-6 Word 2010 中段落设置包含换行选项和字符间距选项,文本对齐方式。由于换行选项和字符间距之间相互影响比较小,在此考虑换行设置之间的相互影响。换行设置共有三个选项,如图 4-8 所示。

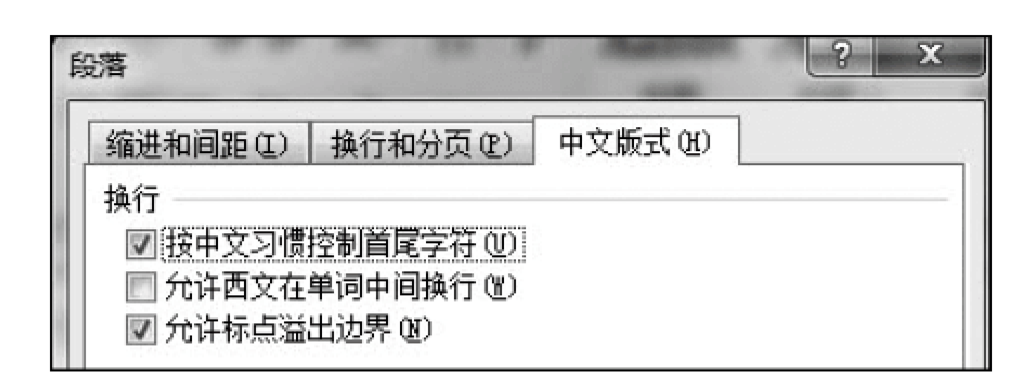


图 4-8 Word 2010 中段落中换行选项

(1) 按中文习惯控制首尾字符;

(2) 允许西文在单词中间换行;

(3) 允许标点溢出边界。

试采用正交法设计测试用例。

解答:分析因素数目和水平数目,该设置共有三个选项:允许中文习惯控制首尾字符、允许西文在单词中间换行、允许标点溢出边界,因素的个数为 3。

每个选项都包含选择和不选择两种情况,则每个因素的水平数相同,均为 2。

选择正交表:

(1) 表中的因数数目大于等于 3;

(2) 表中至少有三个因数的水平数大于等于 2;

(3) 行数取最少的一个。

依据上述原则,选择正交表 $L_4(2^3)$,如图 4-9 所示。

	1	2	3
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

图 4-9 正交表 $L_4(2^3)$

将变量取值和实际含义进行映射如下。

按中文习惯控制首尾字符:0⇒ 不选择;1⇒选择。

允许西文在单词中间换行:0⇒ 不选择;1⇒选择。

允许标点溢出边界:0⇒ 不选择;1⇒选择。

可以得到最后的测试用例,如表 4-4 所示。

表 4-4 WPS 字符设置的测试用例

	按中文习惯控制首尾字符	允许西文在单词中间换行	允许标点溢出边界
测试用例 1	不选择	不选择	不选择
测试用例 2	不选择	选择	选择
测试用例 3	选择	不选择	选择
测试用例 4	选择	选择	不选择

例 4-7 某手机相机软件，其设置如图 4-10 所示。





图 4-10 手机相机的设置界面

共有 6 个设置项：使用音量键作为、自拍、闪光灯、拍摄模式、效果、场景模式。各个选项候选值如表 4-5 所示。

表 4-5 某照相机的设置参数以及取值

编号	选项名称(因素)	候选值(水平)	候选值个数
1	使用音量键作为	缩放键、摄像头键	2
2	自拍	开、关	2
3	闪光灯	开、关	2
4	拍摄模式	正常拍摄、面部优选、全景拍摄	3
5	效果	无效果、复古、黑白	3
6	场景模式	无、肖像、风景、运动、宴会/室内、海滩/雪景	6

在本软件的设置中，设置的选项为 6 个。使用音量键作为、自拍、闪光灯三个因素的水平数为 2，拍摄模式和效果两个因素水平数为 3，场景模式这个因素的水平数为 6。查阅相关的正交表，适合的有 $L_{49}(7^8)$、$L_{18}(3^6\times6^1)$ 等。选择行数较少的 $L_{18}(3^6\times6^1)$ 生成测试用例，如图 4-11(a)所示。

(a)

0	0	0	0	0	0	0
0	0	1	1	2	2	1
0	1	0	2	2	1	2
0	1	2	0	1	2	3
0	2	1	2	1	0	4
0	2	2	1	0	1	5
1	0	0	2	1	2	5
1	0	2	0	2	1	4
1	1	1	1	1	1	0
1	1	2	2	0	0	1
1	2	0	1	2	0	3
1	2	1	0	0	2	2
2	0	1	2	0	1	3
2	0	2	1	1	0	2
2	1	0	1	0	2	4
2	1	1	0	2	0	5
2	2	0	0	1	1	1
2	2	2	2	2	2	0

(b)

0	0	0	0	0	0	0
0	0	1	1	2	2	1
0	1	0	2	2	1	2
0	1	0	0	1	2	3
0	0	1	2	1	0	4
0	1	1	1	0	1	5
1	0	0	2	1	2	5
1	0	0	0	2	1	4
1	1	1	1	1	1	0
1	1	1	2	0	0	1
1	0	0	1	2	0	3
1	1	1	0	0	2	2
0	0	1	2	0	1	3
1	0	0	1	1	0	2
0	1	0	1	0	2	4
1	1	1	0	2	0	5
0	0	0	0	1	1	1
1	1	1	2	2	2	0

图 4-11　手机照相机设置对应的正交表

在该正交表中共有 7 个因素，在实际系统中仅有 6 个因素。如果将照相机的第 1 个因素到第 5 个因素分别和正交表中的第 1 个因素至第 5 个因素对应，正交表第 6 列因素为多余因素，其取值对于相机系统的测试没有作用，可以忽略。

在前三个因素中，被测系统的选项仅有两个，在正交表中提供了三个水平，若第一个选项和第二个选项，分别用 0 和 1 表示，在正交表中前三列中 2 在被测系统中并没有任何含义。为了充分利用这些没有含义的 2，可以将其转换成其有效的选项。在本例中，循环使用前两个选项作为测试用例值，以增加发现问题的能力，如图 4-11(b)所示。

依据正交表的值和系统实际选项进行映射，如表 4-6 所示。

表 4-6　照相机设置参数及其取值

编号	选项名称(因素)	候选值(水平)	水平编号
1	使用音量键作为	缩放键	0
		摄像头键	1

续表

编号	选项名称(因素)	候选值(水平)	水平编号
2	自拍	关	0
		开	1
3	闪光灯	关	0
		开	1
4	拍摄模式	正常拍摄	0
		面部优选	1
		全景拍摄	2
5	效果	无效果	0
		复古	1
		黑白	2
6	场景模式	无	0
		肖像	1
		风景	2
		运动	3
		宴会/室内	4
		海滩/雪景	5

依据映射表和正交表,形成最终的测试用例,如表 4-7 所示。

表 4-7 照相机设置的测试用例

用例编号	使用音量键作为	自拍	拍摄模式	闪光灯	效果	场景模式
1	缩放键	关	关	正常拍摄	无效果	无
2	缩放键	关	开	面部优选	黑白	肖像
3	缩放键	开	关	全景拍摄	复古	风景
…	…	…	…	…	…	…
17	缩放键	关	关	正常拍摄	复古	肖像
18	摄像头键	开	开	全景拍摄	复古	无

4.3 组合测试的数学基础和定义

从 n 个不同元素中,任取 m(m 与 n 均为自然数,下同)个元素按照一定的顺序排成一列,叫作从 n 个不同元素中取出 m 个元素的一个排列;从 n 个不同元素中取出 m 个元素的所有排列的个数,称为从 n 个不同元素中取出 m 个元素的排列数。

$$P_n^m = \frac{n!}{(n-m)!}$$

假设有 4 个自然数，分别用 1,2,3,4 表示，先从中取三个数字并将其排列成一排，共有多少种排列结果？首先从 4 数中取 1,2,3 共三个数，其排列情况为(1,2,3)(1,3,2)(2,1,3)，(2,3,1)，(3,1,2)，(3,2,1)，共 6 种排列。其他以此类推，可以分别取 1,2,4 和 1,3,4 以及 2,3,4 三组数据，每组分别有 6 种排列，共有 24 种情况，及 $P_4^3=4!/(4-3)!=24$。

从 n 个不同元素中，任取 $m(m \leqslant n)$ 个元素并成一组，称为从 n 个不同元素中取出 m 个元素的一个组合。如果两个组合中的元素完全相同，不管元素的顺序如何，都是相同的组合；只有当两个组合中的元素不完全相同时，才是不同的组合。从 n 个不同元素中取出 $m(m \leqslant n)$ 个元素的所有组合的个数，称为从 n 个不同元素中取出 m 个元素的组合数：

$$C_n^m = \frac{n!}{m!(n-m)!}$$

为了对一个具有 n 个参数的程序的输入执行 m 路(维)组合测试，需要对于一个 m 路变量(输入参数)的所有取值组合进行测试，假设每一个变量有 r 种取值。覆盖的取值组合数为：

$$r^m C_n^m = r^m \frac{n!}{m!(n-m)!}$$

每一个测试用例具有 n 个参数的一种取值组合，包含 C_n^m 个 m 参数取值组合情况，这是组合测试的基本依据。

例 4-8 若一个问题具有 5 个参数 a,b,c,d,e，每一个参数具有两个取值，假设需要覆盖三路参数的组合，共需要覆盖的取值组合数为：

$$2^3 \times \frac{5!}{3!(5-3)!} = 80$$

而在 5 个参数中取三个参数，共有 10 种变量组合：

abc，*abd*，*abe*，*acd*，*ace*，*ade*，*bcd*，*bce*，*bde*，*cde*

一个测试用例，例如(0,1,0,0,1)覆盖了 $C_5^3=10$ 种参数的组合，如表 4-8 所示。

表 4-8 (0,1,0,0,1)覆盖的组合情况

1	2	3	4	5	6	7	8	9	10
abc	*abd*	*abe*	*acd*	*ace*	*ade*	*bcd*	*bce*	*bde*	*cde*
010	000	011	001	001	001	100	101	101	001

设待测软件(Software Under Testing，SUT)的因素个数为 n，这些因素数形成有限集合 $P=\{p_1,p_2,\cdots,p_n\}$，其中每个因素 p_i 经过前期处理后共包含 a_i 个取值。为了讨论的方便，将 p_1 的可能取值记为 V_1，p_2 的可能取值集合记为 V_2，以此类推。

若一个 SUT 中所有的参数(因素)的取值数量均相等 $a_1=a_2=\cdots=a_n$，则称该 SUT 为单一水平系统。

若一个 SUT 中的参数(因素)的取值数量不完全相等，即存在 $a_i \neq a_j (i=j)$，则称该 SUT 为混合水平系统。

测试用例：由因数各取一个值，这些值构成的一个元组称为组合测试的一个测试用例。$T_i=(v_{i1},v_{i2},v_{i3},\cdots,v_{in})$，其中 $v_{i1}\in V_1$，$v_{i2}\in V_2$，以此类推。

测试用例集：根据一定的覆盖准则设计的测试用例的集合。

依据测试用例集对于组合覆盖的强度，可以分为单一选择测试、基本选择测试、二维组合测试或者成对测试、N 维组合测试等。

依据覆盖的强度是否相等，可以分为等强度组合测试和变强度测试。

依据是否附带约束条件，可以区分成不带约束的组合测试和带约束的组合测试。

单一选择测试：一个 SUT 中所有因素的所有可能取值至少被测试集中的一个测试用例覆盖。该覆盖准则由 Ammann 和 Offutt 提出。

在单一选择测试中，最少的测试用例个数由所有因素中取值最多的因素的取值个数所决定，即：

$$N_{\min}=\max(|V_i|)\quad 1\leqslant i\leqslant n$$

满足单一选择的测试用例可以使用以下方法产生：从所有因素中，随机选择可能的取值，构成第一个用例。在所有参数的余下取值中各取一个值构成新的测试用例。若某一个参数的所有取值均已经被使用，而其他参数还存在未使用的取值，则该参数可以在已经使用过的取值中取一个，构成新的测试用例。

例 4-9 一个系统有三个输入参数 $P=\{p_1,p_2,p_3\}$，三个参数对应的候选值分别为 $V_1=(a,b)$，$V_2=(1,2,3)$，$V_3=(x,y)$。设计满足单一选择准则的测试用例，如图 4-12 所示。

p_1	p_2	p_3
a	1	x
b	2	y
—	3	—

(a)

p_1	p_2	p_3
a	1	x
b	2	y
a	3	x

(b)

图 4-12 满足单一选择准则的测试用例

首先在三个输入参数的候选取值中，各取一个参数取值，例如，$(a,1,x)$作为第一个测试用例。在此基础上要产生新的用例，对于 p_1 和 p_3 而言，取值分别只剩下 a 和 y，所以直接选取这两个值，而 p_2 可以在 2 和 3 中选取一个值，暂且选择了 2。然后确定第三个测试用例，p_2 只能选择值 3，而 p_1 和 p_3 待定。这样第一列中前两行覆盖了参数 p_1 中的两个参数 a，b。而第二列覆盖了参数 p_2 中的两个参数 1，2，3 三个参数，第三列中前两行覆盖了参数 p_3 中的两个参数 x，y，如图 4-12(a)所示。p_1 的第三行选择 a 和 b 可以任意，或者根据需求中的其他线索决定。类似地，p_3 的第三行选择 x 和 y 可以任意，或者根据需求中的其他线索决定，如图 4-12(b)所示。

显然单一选择测试机械地让每一个参数取值出现一次，并没有考虑信息语义关系，产生的测试用例数目相对较少，其发现缺陷的能力相对较弱。

基本选择测试：一个 SUT 中的测试用例集每次仅由一个因素的可能选项变化而构成。

基本选择测试可以通过如下方法生成测试用例：第一个用例可以随机生成，或者根

据其他信息(如参数取值的重要性或者取值出现的频率)而决定。第二个用例根据第一个参数的取值变化而其他参数值保持初始值不变。在第一个参数取完了所有的值以后,再根据第二个参数的取值变化而产生测试用例,以此类推。在上述所有的变化过程中,不涉及变化的参数均保持初始取值不变。

例 4-10 一个系统有三个输入参数 $P=\{p_1,p_2,p_3\}$,三个参数对应的候选值分别为 $V_1=(a,b)$,$V_2=(1,2,3)$,$V_3=(x,y)$。设计满足基本选择准则的测试用例。

p_1	p_2	p_3
a	1	x
b	1	x
a	2	x
a	3	x
a	1	y

在这个例子中,被测问题和前一个例子相同,但是覆盖的准则不同。首先在第一个测试用例中,各个参数分别选择了 a,1,x,第二个测试用例根据参数 p_1 的取值变化,取值 b,而其他两个参数值 p_2,p_3 保持不变。第 3,4 个测试用例,保持参数 p_1 的初始值 a 不变,根据 p_2 的取值变化而产生两个测试用例。最后一个测试用例仅变化 p_3 的取值。

在基本选择组合过程中,初始测试用例可以随机生成,或者根据其他信息(如参数取值的重要性或者取值出现的频率)而决定。但是初始测试用例的选择将影响不同参数取值出现的频率。除了初始取值以外,其他取值出现的可能性均为 1。因此初始参数选择就显得非常重要。在配置测试中,可以将使用频度最高的参数取值作为初始参数值。在函数测试中,根据经验,估计其出错可能性最大的参数取值作为初始参数值。

在例子中,初始测试用例的选择为$(a,1,x)$,除了初始选择以外的取值,例如参数 p_1 中的 b,参数 p_2 中的 2,3 以及 p_3 中的 y 均只出现了一次。而参数在初始测试用例中的取值出现的次数是:

$$N_i=(|V_1|-1)+(|V_2|-1)+\cdots(|V_{i-1}|-1)+(|V_{i+1}|-1)\cdots|V_n|-1)+1$$

参数 p_1 中 a 的取值出现$(3-1)+(2-1)+1=4$ 次。参数 p_2 中 a 的取值出现了$(2-1)+(2-1)+1=3$ 次。在单一选择中,每次可以在一次用例中同时变化多个参数的选择,而基本选择组合测试中,每次只能变化一个参数的取值,显然满足基本选择的测试用例集,必然满足单一选择准则。

4.4 成对组合测试用例的生成策略

成对测试:对于一个软件的任意两个参数,它们的任意一对有效取值至少被一个测试用例所覆盖。

例如,一个系统有三个输入参数 $P=(p_1,p_2,p_3)$,其中,p_1 的取值 $V_1=\{a,b\}$,p_2 的取值 $V_2=\{1,2,3\}$,p_3 的取值 $V_3=\{x,y\}$。则下面的测试用例满足成对覆盖准则。

p_1	p_2	p_3
a	1	x
a	2	x
a	3	x
a	——	y
b	1	y
b	2	y
b	3	y
b	——	x

在这个测试用例中的三个参数 p_1, p_2, p_3，对于每两个参数构成的任意一对有效取值，(p_1, p_2)、(p_1, p_3)、(p_2, p_3)构成的取值对，如：

p_1	p_2
a	1
a	2
a	3
b	1
b	2
b	3

p_1	p_3
a	x
a	y
b	x
b	y

p_2	p_3
1	x
1	y
2	x
2	y
3	x
3	y

对于其中任意两个因素 p_i 和 p_j 而言，构成的测试用例个数为$|V_i| \times |V_j|$，例如 p_1 和 p_2 两个参数组合数为 6 种。但是对于超过两个因素，满足成对覆盖准则的测试用例个数是不确定的。人们一直采用各种策略实现满足成对测试需求，但是生成的测试用例个数尽可能的少，包括 CATS、AETG、IPO、GA 等。

4.4.1 CATS 算法

CATS(Constrained Array Test System)算法由 Sherwood 提出，采用一次生成一个测试用例，最终生成所需要的组合测试用例的方法。在该方法中，先将所有参数取值构成全组合并用集合 Q 表示，同时构造两个参数的组合放在 UC 集合中。从全组合集合 Q 中选取当前覆盖最多成对组合的组合作为测试用例。

在前面已经讨论过，一个测试用例可能会覆盖多个两参数的组合，不同的用例其覆盖的个数也不相同。例如，有 4 个参数的测试用例，为简单起见，其取值都是{1,2,3,4}，而 UC 当前两两组合集合为{$(p_1=2, p_2=3)$，$(p_1=2, p_3=1)$，$(p_2=3, p_4=4)$，$(p_3=1, p_4=4)$}，则测试用例(2,3,1,4)覆盖两个组合，而测试用例(1,4,1,4)仅覆盖 UC 中的一个组合。

算法 4-1　CATS 算法

(1) TS⇐∅；

(2) UC⇐因素数 P 的成对组合，表示未被覆盖的组合的集合；

(3) Q⇐所有参数取值构成全组合；

(4) while UC≠∅do

① 选择覆盖 UC 中当前最多成对组合的组合作为测试用例，如果存在多个，选择第一个遇到的组合作为测试用例；

② 从 UC 中移去 TC 覆盖的组合；

③ 从 Q 中移去选择作为测试用例的组合；

④ TS⇐TS+TC；

(5) end while

现举例说明 CATS 算法过程。假设一个系统有 4 个参数 p_1,p_2,p_3,p_4，其中第一个参数 p_1 有 4 种取值即 $V_1=\{a_1,a_2,a_3,a_4\}$，p_2 和 p_3 有两种取值，$V_2=\{b_1,b_2\}$，$V_3=\{c_1,c_2\}$，p_4 有三种取值，$V4=\{d_1,d_2,d_3\}$，那么 UC 共有 $4\times2+4\times2+4\times3+2\times2+2\times3+2\times3=44$ 种取值。测试用例集合 TS=∅，Q 的初始值包含所有参数的组合，$Q=\{(a_1,b_1,c_1,d_1),\cdots,(a_4,b_2,c_2,d_3)\}$。UC 包含所有两个参数取值构成的组合。

$$
\begin{aligned}
\text{UC}=\{&(a_1,b_1),(a_1,c_1),(a_1,d_1),(a_1,b_2),(a_1,c_2),(a_1,d_2),(a_1,d_3),\\
&(a_2,b_1),(a_2,c_1),(a_2,d_1),(a_2,b_2),(a_2,c_2),(a_2,d_2),(a_2,d_3),\\
&(a_3,b_1),(a_3,c_1),(a_3,d_1),(a_3,b_2),(a_3,c_2),(a_3,d_2),(a_3,d_3),\\
&(a_4,b_1),(a_4,c_1),(a_4,d_1),(a_4,b_2),(a_4,c_2),(a_4,d_2),(a_4,d_3),\\
&(b_1,c_1),(b_1,d_1),(b_1,c_2),(b_1,d_2),(b_1,d_3),\\
&(b_2,c_1),(b_2,d_1),(b_2,c_2),(b_2,d_2),(b_2,d_3),\\
&(c_1,d_1),(c_1,d_2),(c_1,d_3),\\
&(c_2,d_1),(c_2,d_2),(c_2,d_3)\}
\end{aligned}
$$

开始时，一个测试用例最多可以覆盖 C_4^2 个两参数值组合。第一个取值组合为 $\text{TC}_1=(a_1,b_1,c_1,d_1)$，其覆盖了 $C=\{(a_1,b_1)、(a_1,c_1)、(a_1,d_1)、(b_1,c_1)、(b_1,d_1)、(c_1,d_1)\}$，共 6 个值组合，已经覆盖最多组合的测试用例。所以 $Q=Q-\{(a_1,b_1,c_1,d_1)\}$，$\text{UC}=\text{UC}-C$。

$$
\begin{aligned}
\text{UC}=\{&(a_1,b_2),(a_1,c_2),(a_1,d_2),(a_1,d_3),\\
&(a_2,b_1),(a_2,c_1),(a_2,d_1),(a_2,b_2),(a_2,c_2),(a_2,d_2),(a_2,d_3),\\
&(a_3,b_1),(a_3,c_1),(a_3,d_1),(a_3,b_2),(a_3,c_2),(a_3,d_2),(a_3,d_3),\\
&(a_4,b_1),(a_4,c_1),(a_4,d_1),(a_4,b_2),(a_4,c_2),(a_4,d_2),(a_4,d_3),\\
&(b_1,d_1),(b_1,c_2),(b_1,d_2),(b_1,d_3),\\
&(b_2,c_1),(b_2,d_1),(b_2,c_2),(b_2,d_2),(b_2,d_3),\\
&(c_1,d_2),(c_1,d_3),\\
&(c_2,d_1),(c_2,d_2),(c_2,d_3)\}
\end{aligned}
$$

第二次选择了测试用例 $\text{TC}_2=(a_1,b_2,c_2,d_2)$，第三次选择了测试用例 $\text{TC}_3=(a_2,b_1,c_2,d_3)$，$\text{TC}_4=(a_3,b_2,c_1,d_3)$，这 4 个测试用例其覆盖的两参数组合数均为 6。CATS 算法经过 4 次循环以后，产生了 4 个测试用例，测试用例集合为：

p_1	p_2	p_3	p_4
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_1	c_2	d_3
a_3	b_2	c_1	d_3

UC 的成员成为：

$$
\begin{aligned}
\text{UC}=\{&(a_1,d_3),\\
&(a_2,c_1),(a_2,d_1),(a_2,b_2),(a_2,d_2),\\
&(a_3,b_1),(a_3,d_1),(a_3,c_2),(a_3,d_2),\\
&(a_4,b_1),(a_4,c_1),(a_4,d_1),(a_4,b_2),(a_4,c_2),(a_4,d_2),(a_4,d_3),\\
&(b_1,d_2),\\
&(b_2,d_1),\\
&(c_1,d_2),\\
&(c_2,d_1)\}
\end{aligned}
$$

此时，覆盖 UC 中值对最多的测试用例分别为(a_4,b_1,c_1,d_2)和(a_4,b_2,c_2,d_1)，这两个用例覆盖的 UC 中元素数量都是 5。选择 $TC_5=(a_4,b_1,c_1,d_2)$和 $TC_6=(a_4,b_2,c_2,d_1)$作为测试用例。测试用例集合为：

p_1	p_2	p_3	p_4
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_1	c_2	d_3
a_3	b_2	c_1	d_3
a_4	b_1	c_1	d_2
a_4	b_2	c_2	d_1

UC 的元素为：

$$
\begin{aligned}
\text{UC}=\{&(a_1,d_3),\\
&(a_2,c_1),(a_2,d_1),(a_2,b_2),(a_2,d_2),\\
&(a_3,b_1),(a_3,d_1),(a_3,c_2),(a_3,d_2),\\
&(a_4,d_3)\}
\end{aligned}
$$

此时，覆盖 UC 中值对最多的测试用例分别为(a_2,b_2,c_1,d_1)和(a_3,b_1,c_2,d_1)，这两个用例覆盖的 UC 中元素数量都是 3。选择 $TC_5=(a_2,b_2,c_1,d_1)$和 $TC_6=(a_3,b_1,c_2,d_1)$作为测试用例。测试用例集合为：

p_1	p_2	p_3	p_4
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_1	c_2	d_3
a_3	b_2	c_1	d_3
a_4	b_1	c_1	d_2
a_4	b_2	c_2	d_1
a_2	b_2	c_1	d_1
a_3	b_1	c_2	d_1

此时 UC 包含剩下的 4 个两参数组合：

$$\text{UC}=\{(a_1,d_3),(a_2,d_2),(a_3,d_2),(a_4,d_3)\}$$

由于 4 个两参数组合都是由 p_1 和 p_4 的取值构成，不包含 p_2 和 p_3 的取值，并且取值

之间没有任何的交集，所以增加4个用例，分别包含剩下的 p_1 和 p_4 的取值，如图4-13(a)所示。p_2 和 p_3 的取值可以任意，根据出现的先后次序，也可以根据其他的启发式策略分别加以填充。最后生成的测试用例TS如图4-13(b)所示。

(a)

p_1	p_2	p_3	p_4
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_1	c_2	d_3
a_3	b_2	c_1	d_3
a_4	b_1	c_1	d_2
a_4	b_2	c_2	d_1
a_2	b_2	c_1	d_1
a_3	b_1	c_2	d_1
a_1	—	—	d_3
a_2	—	—	d_2
a_3	—	—	d_2
a_4	—	—	d_3

(b)

p_1	p_2	p_3	p_4
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_1	c_2	d_3
a_3	b_2	c_1	d_3
a_4	b_1	c_1	d_2
a_4	b_2	c_2	d_1
a_2	b_2	c_1	d_1
a_3	b_1	c_2	d_1
a_1	b_1	c_1	d_3
a_2	b_1	c_2	d_2
a_3	b_2	c_1	d_2
a_4	b_2	c_2	d_3

图4-13 测试用例

4.4.2 AETG法

AETG(Automatic Efficient Test Generator)法也属于一次生成一个测试用例的策略，在生成过程中引入随机技术的启发式组合，由D. M. Cohen提出。和前面算法的差异是在生成一个测试用例时的策略。首先从UC中选择出现次数最多的参数取值作为该参数取值，然后将其他参数依据随机排列构成参数元组。从第二个参数开始，一次选择和前面参数构成覆盖最多UC元素作为该参数的取值，形成本轮测试用例的候选。这样的过程需要重复 M 遍。在这 M 个候选中，依据其覆盖UC最多元素确定本轮的测试用例。

当一个参数有多个取值覆盖同样多的参数对时，任意选择一个参数值，作为候选的测试用例，这样会产生不一样的候选测试用例。当 M 个候选的测试用例都已经产生以后，选择其中覆盖UC中最多的参数值对的候选测试用例作为下一个测试用例。在AETG算法中，最后生成的测试用例集规模的大小和 M 值的大小有关。一般地，M 值越大，产生的测试用例数越少。但是当 M 超过50以后，最后测试用例集规模的下降就不明显了。

算法4-2 AETG算法

(1) TS⇐∅;

(2) UC⇐因素数 P 的成对组合，表示未被覆盖的组合的集合。

(3) while UC≠∅do

① for $i=0$ to M do

ⅰ. 在UC中找出出现次数最多的参数取值 v_{ij}，并将对应的参数 p_i 设为第一个参数 p_1'

ⅱ. 对剩下的参数进行随机排列，和第一个参数一起构成了参数序列($p_1', p_2', \cdots, p_k'$)

ⅲ. 从 p_2' 开始，p_i' 选取与前面参数($p_1', p_2', \cdots, p_{i-1}'$)取值构成的组合覆盖UC中最多元素作为本次值。形成一个新的测试用例 TC_i

② endfor

TC⇐max(TC_i)

③ 从 UC 中移去 TC 覆盖的组合；

④ TS ⇐ TS+TC；

(4) end while

现举例说明 ATEG 算法的过程。假设一个系统有 4 个参数 p_1,p_2,p_3,p_4，其中第一个因素 p_1 有三种取值即 $V_1=\{a_1,a_2,a_3\}$，p_2 和 p_3 有两种取值，$V_2=\{b_1,b_2\}$，$V_3=\{c_1,c_2\}$，p_4 有三种取值，$V_4=\{d_1,d_2,d_3\}$，那么共有 36 种取值。测试用例集合 TS=∅，Q 的初始值包含所有参数的组合，$Q=\{(a_1,b_1,c_1,d_1),\cdots,(a_3,b_2,c_2,d_3)\}$。

假设经过三轮循环以后，产生了三个测试用例：

p_1	p_2	p_3	p_4
a_1	b_2	c_1	d_2
a_2	b_2	c_1	d_3
a_3	b_1	c_2	d_1

UC 所包含的集合为：

$$
\begin{aligned}
\mathrm{UC}=\{&(a_1,b_1),(a_1,d_1),(a_1,c_2),(a_1,d_3),\\
&(a_2,b_1),(a_2,d_1),(a_2,c_2),(a_2,d_2),\\
&(a_3,c_1),(a_3,b_2),(a_3,d_2),(a_3,d_3),\\
&(b_1,c_1),(b_1,d_2),(b_1,d_3),\\
&(b_2,d_1),(b_2,c_2),\\
&(c_2,d_2),(c_2,d_3)\}
\end{aligned}
$$

在选择新的候选用例时，计算在 UC 中的出现次数最多的参数值。在上述 UC 中，b_1 和 c_2 均出现 5 次。若选择参数值 c_2，那么将会产生用例模板：

$$(-,-,c_2,-)$$

对于除了 p_3参数以外的其他三个参数做随机排序，假设为 p_4、p_1 和 p_2。首先考虑参数 p_4 的取值情况，$(-,-,c_2,d_1)$没有覆盖 UC 中的任何参数对。而$(-,-,c_2,d_2)$和$(-,-,c_2,d_3)$各覆盖了一个参数值对。假设算法选择 d_2，那么产生了：

$$(-,-,c_2,d_2)$$

接下来选择参数 p_1 的取值情况，$(a_1,-,c_2,d_3)$和$(a_3,-,c_2,d_3)$各覆盖了 UC 中的一个参数对，而$(a_2,-,c_2,d_3)$覆盖了 UC 中的两个参数对，因此参数 p_1 选择值 a_2，形成：

$$(a_2,-,c_2,d_2)$$

最后考虑参数 p_2 的取值情况，(a_2,b_2,c_2,d_3)覆盖了一个参数值对，而(a_2,b_1,c_2,d_3)覆盖了 UC 中的两个参数值对。至此，产生了第一个候选的测试用例：

$$(a_2,b_1,c_2,d_2)$$

这样的过程需要重复 M 遍。在这 M 个候选中，依据其覆盖 UC 最多元素确定本轮的测试用例。

其他步骤和 CATS 方法相同，不再重述。

4.4.3 IPO 法

CATS、AETG 等算法均采用一次生成一个测试用例策略，而 IPO(In Parameter

Order)算法考虑复用已有的测试用例集。IPO 由 Y. Lei 等人提出,其基本思想是首先生成任意两个参数的所有组合,然后依次进行水平扩展和垂直扩展。水平扩展即每次新加入一个参数,并确定其取值;在水平扩展之后,通过垂直扩展来补充尚未被覆盖的配对组合。它与 AETG 算法的根本区别是它不是同时考虑所有参数,而是每次渐进考虑一个参数。该方法依据前两个参数生成满足成对覆盖准则的测试用例集,然后通过扩展测试用例使之能够满足三个参数的成对组合覆盖,以此类推,直至所有的参数都包括到测试用例中,详细描述见算法 4-3。

算法 4-3　IPO 法

输入:系统 S 的参数及其对应的取值集合。

输出:满足成对测试的测试用例集。

(1) TS=$\{(v_i,v_j)\mid v_i\in V_1, v_j\in V_2\}$;　　%取第一个和第二个因素的组合

(2) $n=|P|$　　%n 为参数个数

(3) if $n==2$ then 算法终止。

(4) for $p_i(i=2,3,\cdots,n)$ do

水平扩展 IPO_H(TS,p_i)

for each test$(v_1,v_2,\cdots,v_{i-1})$ in TS do

用$(v_1,v_2,\cdots,v_{i-1},v_i)$替换$(v_1,v_2,\cdots,v_{i-1})$,　%根据不同的策略选择 v_i

endfor

垂直扩展 IPO_V(π,TS);

%π 是由参数 $p_1,p_2,\cdots,p_i$ 取值构成的,未被 TS 覆盖的值对。

While TS 未覆盖 p_i 和 $p_1,p_2,\cdots,p_{i-1}$ 构成的值对 do

增加针对参数 $p_1,p_2,\cdots,p_{i-1},p_i$ 的一个新的用例到 TS;

end while

(5) endfor

Y. Lei 等人分析 IPO 算法的优点在于:

(1) IPO 策略允许在水平扩展和垂直扩展时采用不同的优化策略来生成测试用例。

(2) 在软件更新、接口增加新参数时复用已有的测试用例。假设对于系统 S,已经生成的测试用例为 TS,新版本为 S'。通过 IPO 策略,将通过扩展 TS 来产生新的测试用例 TS′,可以节省产生测试用例的精力。

(3) 在软件更新,若某一个参数增加新的取值时,复用已有的测试用例。假设对于系统 S,已经生成的测试用例为 TS,新版本为 S'。在这种情况下,仅应用垂直扩展增加新的用例,即可生成新测试用例集 TS′。TS′的测试用例集,可能比重新生成新的用例集的数量更多些,但节省了比较多的精力。

通过不同的策略可以有不同的水平扩展算法,事实上 Y. Lei 本人就已经提出了 IPO_H_IV 算法和 IPO_H_EC 算法。本书仅介绍其中的 IPO_H_IV 水平扩展。首先比较当前参数 p_i 的取值个数$|V_i|$和已经生成的测试用例个数$|TS|$大小,将其最小值设为 s。对于参数 p_i 而言,前 s 个参数,不需要做任何的启发式动作,直接将其扩展到测试用例中;

若已经生成的测试用例个数多于参数个数，则对于余下的测试用例的扩展，选择其覆盖参数值对最多的参数值作为 p_i 的取值；若 p_i 的取值个数多于已经生成的测试用例个数，则直接为 p_i 的前 s 个取值，p_i 多余的取值在垂直扩展中增加。其详细描述见算法 4-4。

算法 4-4　IPO_H_IV 算法

输入：由参数 $p_1, p_2, \cdots, p_{i-1}$ 构成的测试集 TS，水平扩展参数 p_i。

输出：增加 p_i 取值水平扩展以后的测试集 TS。

%初始化；

(1) π ={参数 p_i 和参数 $p_1, p_2, \cdots, p_{i-1}$ 之间构成的值对}

　　%s 为 p_i 参数值个数和已经产生的测试用例个数的最小值。

(2) $s=\min(|V_i|, |TS|)$　　　%简单选择 p_i 的前 s 个值。

(3) for $j=1$ to s do

　给参数 p_i 赋值 v_{ij} 扩展第 j 个测试用例；

　从 π 中移除覆盖的值组合。

(4) end for

(5) if $|TS|>|V_i|$ then

　　%已经产生测试用例个数大于参数 p_i 的取值个数 $|V_i|$

　① for $j=|V_i|+1$ to $|TS|$ do

　　选择参数 p_i 中覆盖 π 中组合最多的取值来扩展第 j 个测试用例；

　　从 π 中移去已覆盖的组合。

　② end for

(6) end if

通过算法 4-4 水平扩展以后，如果还存在未被覆盖的值对，那么执行算法 4-5 进行扩展。

算法 4-5　IPO_V_IV 算法

输入：水平扩展后的测试用例集 TS，尚未覆盖的组合集 π。

输出：完成 p_i 取值垂直扩展的测试用例集。

(1) π ={参数 p_i 和参数 $p_1, p_2, \cdots, p_{i-1}$ 之间构成的、尚未被 TS 覆盖的值对集合}

(2) $TS'=\varnothing$

(3) for (v_{iu}, v_{jw}) in π do

　① if TS′中包含测试用例，其 p_i 的取值为“－”，p_j 的取值为 w then

　　修改该测试用例，将“－”替换成 u；

　② else

　　产生新的测试用例 tc，该用例中 p_i 的取值为 u，p_j 的取值为 w，其他参数的取值为“－”。

　　$TS' = TS' \cup \{tc\}$

　③ endif

(4) end for

(5) $TS=TS \cup TS'$

现举例说明 IPO 算法实现过程。系统 S 具有三个参数 p_1, p_2, p_3，p_1 的取值集合为 $V_1=\{a, b\}$，p_2 的取值集合为 $V_2=\{1, 2\}$，p_3 的取值集合为 $V_3=\{x, y, z\}$，依据 IPO 算法

构建满足两两组合的测试用例集。

依据构建前两个参数 p_1,p_2 的取值两两覆盖准则。前两个参数均具有两个取值，共有4个元素：

p_1	p_2
a	1
a	2
b	1
b	2

接下来，考虑 p_3 的水平扩展，由于参数 p_3 有三个取值，而 p_1 和 p_2 已经产生的参数取值共有4个用例。依据水平扩展算法，前三个测试用例直接取 p_3 的值，产生如下测试用例。

p_1	p_2	p_3
a	1	x
a	2	y
b	1	z
b	2	—

对于第4个测试用例，p_3 值没有确定，可以选择 x,y,z 三个取值中的一个。扩展到第三个参数，构成未被测试用例覆盖的值对包括：$\pi=\{(a,z),(b,x),(b,y),(1,y),(2,x),(2,z)\}$。如果选择 x 作为参数 p_3 的值，那么构成了测试用例 $(b,2,x)$，其覆盖 π 中的两个值对 (b,x) 和 $(2,x)$。选择 y 作为参数 p_3 的值，那么构成测试用例 $(b,2,y)$，其覆盖 π 中一个值对 (b,y)。如果选择 z 作为参数 p_3 的值，那么构成了测试用例 $(b,2,z)$，其覆盖 π 中的一个值对 $(2,z)$。因此，可以选择 x 作为参数 p_3 的值，那么构成了测试用例 $(b,2,x)$，如下所示：

p_1	p_2	p_3
a	1	x
a	2	y
b	1	z
b	2	x

自此，完成 p_3 参数的横向扩展，已经生成的测试用例集中共有4个测试用例，未被覆盖的值对包括：$\pi=\{(a,z),(b,y),(1,y),(2,z)\}$。接下来执行垂直扩展来覆盖 π 中的值对。在开始时 $TS'=\varnothing$。对于 π 中的 (a,z) 的值对，(a,z) 被测试用例 $(a,-,z)$ 所覆盖，这里－表示在当前暂时不关心其取值。将其增加到 TS' 中，$TS'=\{(a,-,z)\}$。接下来考虑第二个值对 (b,y)，(b,y) 中的 b 和 y 均未在 TS' 测试用例中出现过，所以产生新的测试用例 $(b,-,y)$，并将其添加到 TS' 中，$TS'=\{(a,-,z),(b,-,y)\}$。接下来考虑 $(1,y)$，由于 y 在 TS' 中出现过，并且其 p_2 对应的值为“－”，所以将 TS' 中的测试用例 $(b,-,y)$ 修改为 $(b,1,y)$。对于第4个值对 $(2,z)$，由于在 TS' 中存在测试用例 $(a,-,z)$，存在 z，并且 p_2 值为“－”，所以将其 $(a,-,z)$ 修改为 $(a,2,z)$，所以 $TS'=\{(a,2,z),(b,1,y)\}$，$TS=TS\cup TS'$。

p_1	p_2	p_3
a	1	x
a	2	y
b	1	z
b	2	x
a	—	z

⇒

p_1	p_2	p_3
a	1	x
a	2	y
b	1	z
b	2	x
a	—	z
b	—	y

⇒

p_1	p_2	p_3
a	1	x
a	2	y
b	1	z
b	2	x
a	—	z
b	1	y

⇒

p_1	p_2	p_3
a	1	x
a	2	y
b	1	z
b	2	x
a	2	z
b	1	y

重复上述过程对因素集合中剩余的因素进行水平扩充和垂直扩充可得到最终组合覆盖表，若表中仍有未填的值“—”，则从该因素可选值集合中随机选择产生一个值替换“—”。

4.4.4 GA 法

构建所有参数覆盖组合 UC。然后从一个空的测试用例集开始，每次利用遗传算法增加一个测试用例到测试用例集中，并删除该测试用例在 UC 覆盖的参数组合，一直到 UC 为空。

遗传算法是一类模拟生物进化的智能优化算法，它是由 J. H. Holland 于 20 世纪 60 年代提出。通过模拟自然界并应用随机理论而形成的生命进化机制，其主要特征在于群体搜索策略和简单的遗传算子，群体搜索可使遗传算法实现整个解空间的分布式信息探索、采集和继承。遗传算法是从种群（代表问题潜在解的集合）开始的，按照适者生存和优胜劣汰的原理，逐代演化产生出越来越好的近似解。在每一代，根据问题域中个体的适应度大小挑选个体，并借助于自然遗传学的遗传算子进行组合交叉和变异，产生出代表新的解集的种群。新生代种群比前代更加适应于环境，末代种群中的最优个体经过解码，可以作为问题近似最优解。遗传算法包括三个基本操作：选择、交叉和变异。

算法的具体描述如下。

(1) 初始化参数组合 UC 和测试用例集 TS。

(2) 利用遗传算法计算当前覆盖 UC 近似最多的测试用例 TC。

(3) 将 TC 添加到集合 TS 中 TS＝TS∪TC，删除 TC 覆盖的 UC 中参数组合。

(4) 重复步骤(2)和(3)，直到 UC 为空。

设系统 S_1 输入由 a、b 和 c 三个参数组成，参数取值个数依次为 2、2 和 3，p_1 的其取值集合 $V_1=\{a,b\}$，p_2 的其取值集合 $V_2=\{1,2\}$，p_3 的其取值集合 $V_3=\{x,y,z\}$，那么可以构造 UC，如表 4-9 所示。

用遗传算法产生组合测试用例，首先要执行测试用例的编码。所谓编码方式，就是将问题的潜在解使用适合遗传算法的基因编码表示。对于组合测试而言，就是将参数映射成二进制编码。由于不同的参数的取值个数并不相同，一个参数在基因编码中的位数也不相同。一个参数 p_i 的可能取值个数为 L_i，如果 $2^{n-1}\leqslant L_i\leqslant 2^n$，那么该参数的编码长度为 n。

表 4-9　GA 算法中 UC 集合示例

第一值对		第二值对		第三值对	
p_1	p_2	p_1	p_3	p_2	p_3
a	1	a	x	1	x
a	2	a	y	1	y
b	1	a	z	1	z
b	2	b	x	2	x
——	——	b	y	2	y
——	——	b	z	2	z

设有一个系统 S_2 输入由 4 个参数组成(p_1,p_2,p_3,p_4),参数取值个数依次为 4,2,2,3。由于组合测试关注点在于测试用例的生成,用参数取值的序号来表示参数的实际值,这样可以避免参数类型的转换。例如,测试用例(3,0,1,2)表示 p_1 取第三个值(第一个为 0),p_2 取第 0 个值,p_3 取第一个值,p_4 取第二个值。若 $L_i \leqslant 2^n$,那么在 L_i 和 2^n 之间随机填充所允许的参数值。系统 S_2 可以采用长度为 6 的二进制编码表示,p_1 占用位 b_0b_1,p_2 占用位 b_2,p_3 占用位 b_3,p_4 占用位 b_4b_5。对于 p_4 而言,前三个编码 00,01,10,分别为 v_{41},v_{42},v_{43},编码 11 可以随机为 v_{41}、v_{42} 和 v_{43} 中间的任意一个,可以使用启发式算法选取,以增加测试效果。最后的编码情况如表 4-10 所示。

表 4-10　参数取值以及编码

参数 p_1		参数 p_2		参数 p_3		参数 p_4	
b_0	b_1	b_2		b_3		b_4	b_5
00	v_{10}	0	v_{20}	0	v_{30}	00	v_{40}
01	v_{11}	1	v_{21}	1	v_{31}	01	v_{41}
10	v_{12}	--		--		10	v_{42}
11	v_{13}	--		--		11	-

利用遗传算法生成的测试用例集,要求在满足成对测试准则的前提下,测试用例数尽可能的少,因此在每一代进化时选择尽可能多覆盖 UC 中参数组合的测试用例。设群体数量为 N,个体(测试用例)TC_i 覆盖的 UC 中参数组合数为 C_i,$1 \leqslant i \leqslant N$,则个体 TC_i 的适应度函数可以表示如下:

$$f_i = \frac{c_i}{\sum_{i=1}^{N} c_i}, \quad \text{其中} \quad \sum_{i=1}^{N} f_i = 1$$

计算当前群体的适应度值后,从当前群体中选择一些个体作为新一代群体的父辈。若个体的适应度高,则被选中的几率较大,且可能多次选中;反之,几率较小,甚至不会被

选中。根据个体(测试用例)的适应度值 f_i 确定选择的概率 $PS_i = f_i$。选择的累积函数 C_j 定义为：

$$C_j = \sum_{i=1}^{j} PS_i$$

在选择新个体时，按照赌轮方式来确定，每个个体 TC_j 处于选择区间 $[C_j, C_{j+1})$。计算时每次产生一个选择随机数，那么处于对应选择区间的个体将被选中作为进化成员。

在选择以后，个体通过交叉和变异实现进化。交叉采用单点交叉的方法实现。两个长度为 N 的个体，分别表示如下：

a_1	a_2	$\cdots$	a_{m-1}	a_m	$\cdots$	a_N
b_1	b_2	$\cdots$	b_{m-1}	b_m	$\cdots$	b_N

在 $m(1 \leqslant m \leqslant N)$ 处交叉，则产生两个新的个体：

a_1	a_2	$\cdots$	a_{m-1}	b_m	$\cdots$	a_N
b_1	b_2	$\cdots$	b_{m-1}	a_m	$\cdots$	b_N

对应于系统 S_2 两个测试用例及其编码为：

$$\begin{matrix} TC_1 = (3,0,1,2) & \Rightarrow & CD_1 = 110110 \\ TC_2 = (1,1,0,2) & & CD_2 = 011010 \end{matrix}$$

假设选择交叉点为 4 位，那么交叉以后的编码和测试用例分别为：

$$\begin{matrix} CD_1' = 110110 & \Rightarrow & TC_1' = (3,0,0,2) \\ CD_2' = 011010 & & TC_2' = (1,1,1,2) \end{matrix}$$

变异可以促进群体的多样化，防止群体进化过早地收敛，即防止群体进化停滞不前或冻结，若无变异，则新群体中的测试数据值即为局限于初始化的数值。变异对于个体选定一个变异位，在变异几率的控制下，将变异位用随机数替换，变异过程将产生单个新个体，添加到新一代群体中。在本算法中采用二进制编码，对于一个个体(测试用例)：只需将变异位取反就可以得到新的测试用例。以测试用例(3,0,1,2)的编码表示为 110110 为例，选择变异位为第二位和第三位，那么变异以后的编码为 101110，对应的测试用例演变成为(2,1,1,2)。

$$TC_1 = (3,0,1,2) \Rightarrow CD_1 = 110110 \Rightarrow CD_1' = 101110 \Rightarrow TC_2' = (2,1,1,2)$$

4.5 可变强度和具有约束的组合测试

在 4.4 节的讨论过程中，组合测试策略都是假设任意的 k 个参数之间存在的交互关系都相同，而实际的软件系统的输入参数之间的关系有如下多种形式。

(1) 有些输入参数之间不存在约束关系；

(2) 有些输入参数之间可能会存在约束关系；

(3) 参数之间存在交互关系，但是不同的参数之间的组合要求强度是不相同的。

4.5.1 混合强度的组合测试

组合方法所产生的测试用例集 TS 是一个 $m\times n$ 的矩阵。其中，n 为被测函数的参数（或者被测配置的因素），m 为测试用例集 TS 所包含的测试用例数量。对于一个正整数 k（$1\leqslant k\leqslant n$），如果 TS 中的任意 k 列，即第 $i_1,i_2,\cdots,i_k$ 列均要求满足 k 路组合，则 k 称为组合强度。其所产生的测试用例集为固定强度组合测试用例套，k 可以称为组合强度。固定强度组合测试可以表示为：

$$\mathrm{TS}_{\mathrm{Inv}}(P,m,k)$$

式中，P 为参数或者因素的集合；m 为测试用例的个数；k 为组合的强度。

若组合强度为 k 的组合测试用例集 TS 存在 t 个子集 $\mathrm{TS}_i\subseteq\mathrm{TS}(t=1,2,\cdots,t)$，其中，$\mathrm{TS}_i$ 包括 n_i 个因素，因素之间构成了 $m\times n_i$ 个元素的矩阵。其中，TS_i 为强度为 k_i 的固定组合强度测试集，且 $k_i\neq k$，则称其为混合强度的组合测试。其具有 n_i 因素子集及其强度 k_i，可以表示如下：

$$C_i=\{p_{i1},p_{i2},\cdots,p_{in_i}\}@k_i$$

混合强度的组合的基本含义，在全局的基础上，对于部分参数的组合提出了更高的要求。若有 A、B、C、D、E 5 个参数，其取值均是 0、1。若强度为 2，其覆盖的组合如表 4-11 所示。

表 4-11 两个参数之间的组合情况

AB	AC	AD	AE	BC	BD	BE	CD	CE	DE
00	00	00	00	00	00	00	00	00	00
01	01	01	01	01	01	01	01	01	01
10	10	10	10	10	10	10	10	10	10
11	11	11	11	11	11	11	11	11	11

若在此基础上，加入 C={B,C,D}@3，那么覆盖的组合如表 4-12 所示。

表 4-12 增加第三个参数强度以后的组合要求

AB	AC	AD	AE	BCD	BE	CE	DE
00	00	00	00	000	00	00	00
01	01	01	01	001	01	01	01
10	10	10	10	010	10	10	10
11	11	11	11	011	11	11	11
--	--	--	--	100	--	--	--
--	--	--	--	101	--	--	--
--	--	--	--	110	--	--	--
--	--	--	--	111	--	--	--

相比强度为 2 的组合测试，其覆盖的两参数组合 BC、BD、CD 均包含在一个三参数组合中。

对于变强度的组合测试，其组合要求，可以在固定强度组合的基础上，添加约束表示，该约束用因素集合及其组合强度表示。

$$C = \{C_1, C_2, \cdots, C_i\}$$

变强度的组合表示为：

$$TS_v(P, m, k, C)$$

式中，P 为参数或者因素的集合；m 为测试用例的个数；k 为全局的组合强度；C 为个性化的强度要求的集合。

例如，一个函数（或者系统）包含 4 个输入参数（配置因素），为了区分各个参数的取值，分别用不同的代号表示，其本身没有特定的含义：

$$V_1 = \{1, 2\}$$

$$V_2 = \{a, b\}$$

$$V_3 = \{I, II\}$$

$$V_4 = \{x, y, z\}$$

若采用一个成对组合测试，可以找到一个用例数为 6 的测试用例集 $TS_{Inv}(P, 6, 2)$。

p_1	p_2	p_3	p_4
1	a	I	x
1	a	II	y
1	b	I	z
2	a	II	z
2	b	I	y
2	b	II	x

若根据系统的特征分析，认为 p_1，p_2 和 p_3 三个参数必须满足强度为 3 的组合测试要求，即增加了一个强度要求：

$$C_1 = \{p_1, p_2, p_3\}@3$$

显然上述测试用例并不能满足要求。一个满足该要求的测试用例表示如下：

p_1	p_2	p_3	p_4
1	a	I	x
1	a	II	y
1	b	I	z
1	b	II	x
2	a	I	y
2	a	II	z
2	b	I	x
2	b	II	y

从目前的情况看，变强度的测试用例生成基本上都是围绕着“一次生成一个用例”的方式来生成，其具体的生成方法不再详细讨论。

4.5.2 参数值之间的约束

在组合测试过程中，特别是系统配置类的测试场景中，不同因素之间存在一定约束关系。这些约束关系可能来自于系统以外，无法通过程序自动处理。例如，当 CPU 为 x86 类型时，其支持的最大内存为 4GB，若选择内存为 64GB 则没有任何意义，其主板也不支持 64GB 的内存。所以当 CPU 选择为 x86 类型时，内存为 64GB 的测试组合为无效测试用例。若不考虑这些约束关系，其所产生的测试用例集包含大量无效测试用例。无效的测试用例，包含一些无效的取值组合，也有可能包含一些有效的取值组合。仅删除无效测试用例，会导致最终的测试用例集不能实现两因素或多因素组合覆盖。因素之间存在约束关系的系统，在生成测试用例前确定约束关系，让组合测试工具根据约束来生成有效的测试用例集。约束关系一般通过条件表达式来实现。

由于不同的工具对约束表达式存在较大差异，下面以微软公司的 PICT 组合测试工具为例来说明约束的表达。例如，一个系统环境，包含 Type、Size、Format method、File system、Cluster size、Compression 等 6 个配置参数，其对应可能的取值用冒号右边以逗号分隔的一系列值来表示。

```
Type:Primary, Logical, Single, Span, Stripe, Mirror, RAID-5
Size:10, 100, 500, 1000, 5000, 10000, 40000
Format method:quick, slow
File system:FAT, FAT32, NTFS
Cluster size:512, 1024, 2048, 4096, 8192, 16384, 32768, 65536
Compression:on, off
```

参数值之间的约束关系一般用条件判断来实现，其一般的规则为：

```
IF  Condition1  THEN  Condition2  ELSE Condition3
```

这里 Condition1、Condition2、Condition3 均为条件表达式或者关系表达式，一般不存在赋值或者其他语句。其含义是如果 Condition1 满足，那么 Condition2 也必须满足，否则 Condition3 必须满足。

例如，FAT 格式的文件大小不能超过 4096，FAT32 格式的文件大小不能超过 32 000，约束关系可以表示为：

```
IF [File system] = "FAT"   THEN [Size]≤4096;
IF [File system] = "FAT32" THEN [Size]≤32000;
```

逻辑表达式，一般由逻辑运算符连接关系表达式而成，逻辑运算符支持 NOT，AND 和 OR 三种运算。其中，NOT 为逻辑非，是单目运算符，其结果和运算对象的逻辑结果相反。AND 为逻辑与运算，是双目运算符，只有参加运算的对象同时为真，结果才为真。OR 为逻辑或运算，是双目运算符，只要参加运算的对象有一个为真，结果就为真。

关系表达式，由关系运算符=，<>，>，>=，<，<=连接不同的变量或常量表达式而成。在 PICT 中，定义了变量或者常量表达式的类型，从而支持关系表达式。PICT 支

持字符串和数值两种类型。除了字符串以外,均认为是数值类型,如表 4-13 所示。

例如:

```
IF [File system] < >  "NTFS" OR  ([File system] = "NTFS" AND [Cluster size]> 4096)
THEN [Compression] =  "Off";
```

个体和集合之间的关系,可以用 IN 表示。

例如:

```
IF [File system] IN {"FAT", "FAT32"} THEN [Compression] = "Off";
```

同时支持通配符: * 表示任何字符,?表示一个字符。

表 4-13 PICT 中各种约束表达式

	设计运算符
条件表达式	IF Condition1 THEN Condition2 ELSE Condition3
逻辑运算符	NOT,AND,OR
关系运算符	=,<>,>,>=,<,<=
通配符	* 表示任何字符,?表示一个字符
集合关系	IN

但在实际应用中,有时参数的表达比较复杂。例如,表 4-14 中给出了一个约束的例子。

表 4-14 复杂关系的约束示例

参数	取值
A	0,1
B	0,1
C	0,1

约束:
IF [A]=0 THEN [B]=0;
IF [B]=0 THEN [C]=0;
IF [C]=0 THEN [A]=1;

在这个约束中,当 A 取值 0 时,B 只能取值 0,而 B 取值为 0 时,C 只能取值 0,而 C 取值 0 时,A 取值为 1,显然这和前面预定义 A 取值 0 矛盾。因此,包含 A 取值为 0 的测试用例均是无效的测试用例,而不能简单去考察 A 和 B 之间的约束值。同时,包含 B 取值为 0,而 C 取值为 1 的测试用例也是无效的测试用例。

增加约束,并不一定减少测试用例。例如,a,b,c 三个参数取值均为 0,1。在没有约束的条件下,产生如下包含 4 个测试用例的测试用例集。其模型文件内容为:

a: 0,1

b: 0,1

c: 0,1

在组合强度为 2 的情况下,PICT 产生的测试用例为:

a	b	c
0	1	0
0	0	1
1	1	1
1	0	0

若加上约束，当 a 因素取值为 0 时，b 因素取值也一定为 0。则其模型文件内容为：

a：0，1

b：0，1

c：0，1

IF $[a]=0$ THEN $[b]=0$；

在强度为 2 的情况下，PICT 产生的测试用例集为：

a	b	c
1	1	1
0	0	1
1	1	0
0	0	0
1	0	0

4.5.3 种子组合和负面测试

前面所讨论的约束，都是对参数取值之间的逻辑关系的约束。在实际应用中，由于有些参数值之间组合的重要性，要求在实际测试过程中，测试必须覆盖到该组合。这种必须测试到的参数组合称为种子组合。一般而言，种子组合用于测试一些关键参数值组合。另一方面，通过种子组合，也可以使得以前产生的测试用例重新得到利用。在 PICT 中，种子组合是一个单独文件。第一行是用 Tab 分隔的参数名称，后继每一行表示一个种子组合，在种子组合中的值之间也是用 Tab 进行分隔。每一行的种子组合可以是完整的，每一个参数均指定了特定的值，也可以是忽略其中的部分参数的取值。

假若有如下的 PICT 模型文件 model. txt：

```
#
#Machine
#
OS:   Win2K, WinXP
SKU:  Pro, Server, Datacenter, WinPowered
LANG: EN, DE
ARCH: X86,IA64
```

一个可能的种子文件如下：

```
OS       SKU      LANG     ARCH
Win2K    Pro      EN       X86
```

```
Win2K            DE     X86
WinXP    Pro     EN     IA64
```

原则上，种子文件的参数、值及其约束关系应该和模型文件中的参数一致。如果不一致，PICT 将根据如下原则处理种子组合。

（1）忽略种子文件中包含模型文件中不存在的参数。

（2）忽略种子文件中在模型文件中不存在的参数值。

（3）忽略种子文件中违反了在模型文件中的定义约束条件的行。

例如，若有一个系统的模型文件如下：

a：a_1，a_2，a_3

b：b_1，b_2

c：c_1，c_2，c_3，c_4

d：d_1，d_2

在没有添加任何种子文件时，利用 PICT 所生成的测试用例集如表 4-15 所示。

表 4-15 未带种子文件的测试用例

编号	a	b	c	d
1	a_1	b_1	c_2	d_2
2	a_1	b_2	c_3	d_1
3	a_2	b_2	c_3	d_2
4	a_2	b_1	c_2	d_1
5	a_1	b_2	c_4	d_2
6	a_2	b_1	c_1	d_1
7	a_3	b_1	c_4	d_1
8	a_1	b_2	c_1	d_2
9	a_2	b_1	c_4	d_1
10	a_3	b_2	c_3	d_2
11	a_3	b_1	c_1	d_1
12	a_1	b_1	c_3	d_2
13	a_3	b_2	c_2	d_2

依据系统特性，将（a_1，b_1，c_1，d_1）增加到种子文件中。即种子文件内容为：

$a \quad b \quad c \quad d$

$a_1 \quad b_1 \quad c_1 \quad d_1$

在 PICT 所产生的测试用例中，其中第一行就是种子文件中所提供的测试组合。在没有增加任何种子时，PICT 所产生的测试用例集中包含 13 个测试用例，而增加了一个组合以后，其测试用例集合反而降低为 12 个用例。由此可以看出增加种子组合，并不一定增加测试用例，如表 4-16 所示。

表 4-16 带有两个种子组合的测试用例

编号	a	b	c	d
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_4	d_2
3	a_3	b_2	c_3	d_1
4	a_3	b_1	c_2	d_2
5	a_1	b_2	c_2	d_1
6	a_2	b_1	c_2	d_2
7	a_2	b_1	c_3	d_1
8	a_1	b_1	c_3	d_2
9	a_2	b_1	c_4	d_1
10	a_2	b_2	c_1	d_2
11	a_3	b_1	c_1	d_2
12	a_3	b_1	c_4	d_2

依据系统特性，将(a_1,b_1,c_1,d_1)和(a_2,b_2,c_2,d_2)增加到种子文件中，即产生的种子文件内容为：

$$\begin{matrix} a & b & c & d \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \end{matrix}$$

在 PICT 所产生的测试用例中，其中前两行就是种子文件中所提供的测试组合，其所产生的测试用例集合为 12 个用例，如表 4-17 所示。

表 4-17　带有三个种子组合的测试用例

编号	a	b	c	d
1	a_1	b_1	c_1	d_1
2	a_2	b_2	c_2	d_2
3	a_3	b_1	c_4	d_2
4	a_2	b_2	c_4	d_1
5	a_2	b_1	c_3	d_1
6	a_3	b_2	c_1	d_1
7	a_3	b_1	c_2	d_1
8	a_1	b_2	c_2	d_2
9	a_1	b_2	c_3	d_2
10	a_3	b_2	c_3	d_1
11	a_1	b_1	c_4	d_1
12	a_2	b_1	c_1	d_2

依据系统特性，将(a_1,b_2,c_3,d_2)和(a_2,b_2,c_4,d_2)增加到种子文件中，即产生的种子

文件内容为：

$$
\begin{array}{cccc}
a & b & c & d \\
a_1 & b_2 & c_1 & d_2 \\
a_1 & b_2 & c_2 & d_2 \\
a_1 & b_2 & c_3 & d_2 \\
a_1 & b_2 & c_4 & d_2
\end{array}
$$

在 PICT 所产生的测试用例中，其中前两行就是种子文件中所提供的测试组合，其所产生的测试用例集合为 13 个用例，如表 4-18 所示。

表 4-18　带有三个种子组合的测试用例

编号	a	b	c	d
1	a_1	b_2	c_1	d_2
2	a_1	b_2	c_2	d_2
3	a_1	b_2	c_3	d_2
4	a_1	b_2	c_4	d_2
5	a_2	b_1	c_4	d_1
6	a_3	b_2	c_3	d_1
7	a_3	b_1	c_4	d_2
8	a_2	b_2	c_1	d_2
9	a_2	b_1	c_2	d_1
10	a_2	b_1	c_3	d_1
11	a_3	b_1	c_2	d_2
12	a_1	b_1	c_1	d_1
13	a_3	b_2	c_1	d_2

在软件测试过程中，经常还需要使用一种称为“负面测试”的技术，其核心本质是输入不合法的数据，用于确认程序是否正确处理了错误的输入。在负面测试过程中，在一个测试用例中，仅允许出现一个参数取值为不合法的值。在一般的程序第 1 次遇到一个错误时，立即进入到异常处理，这样有可能屏蔽后面不合法数据的检测功能。例如，在程序 4-1 中，求变量 a、变量 b 和变量 c 的平方根的和。加入简单输入执行 print sumsquareroot (−4，−9，−16)语句，−4 和−9，−16 均是不合法数据，但是在语句[1]中已经抛了异常，实际上使得语句[2]和语句[3]并没有被测试到。

程序 4-1　一个负面测试例子

```
import math
def sumsquareroot(a,b):
    if (a<0):raise;     #[1]
    if(b<0):raise;     #[2]
```

```
    if (b< 0) :raise;      # [3]
    return (math.sqrt (a)+ math.sqrt (b)+ math.sqrt (c))
```

为了避免这种情况出现，需要在产生组合测试用例之前，确定哪些是非法数据，哪些是合法数据。在最后产生的测试用例中，避免在一个测试用例中同时出现多个非法数据值。在 PICT 中，默认采用在参数值之前增加～来表示。

例如，上述程序中，模型文件内容为：

a：～－1,0,1

b：～－1,0,1

c：～－1,0,1

其产生的测试用例如表 4-19 所示。

表 4-19 带有负面测试的测试用例

a	b	c
0	1	0
0	0	1
1	1	1
1	0	0
0	1	～－1
1	～－1	0
～－1	0	0
1	～－1	1
1	0	～－1
0	～－1	0
～－1	1	1

第 5 章　基于有限状态机的测试

在现实世界中,有许多事情可以用有限个状态来表达,如红绿灯、电话、电梯等。在软件建模过程中,很多系统都是由有限的状态所组成,在不同的状态和输入作用下产生下一个状态。有限状态机已经成为刻画系统状态的重要建模工具,同时也为软件测试提供重要的依据。本章将讨论基于有限状态机的软件测试,包括周游法、区分序列法、特征序列法、唯一输入/输出序列等。

5.1　有限状态机的定义

5.1.1　有限状态机

状态机理论在数字电路设计领域,用于描述电子逻辑电路逻辑的变化。在软件设计领域,状态机的理论用于描述一些复杂的算法、内部的结构和流程,例如通信协议、实时系统、面向对象软件中类的行为及其交互。有限状态机(Finite State Machine,FSM),表示有限个状态以及在这些状态之间的转移和动作的数学模型。因为有限状态机具有有限个状态,所以可以在实际的工程上实现。但这并不意味着其只能进行有限次的处理,相反,有限状态机是闭环系统,可以用有限的状态,处理无穷的事务。

现以电子控制门作为例子来说明状态机。在很多建筑中都有一个自动感应门,该门只允许从外向内单向通行,如图 5-1 所示。为了实现自动控制,在门外面通常会有一个感应区,当有人进入感应区时,门将自动打开。当有人站在开门区时,门禁止关闭以防止门碰到站开门区里的人。

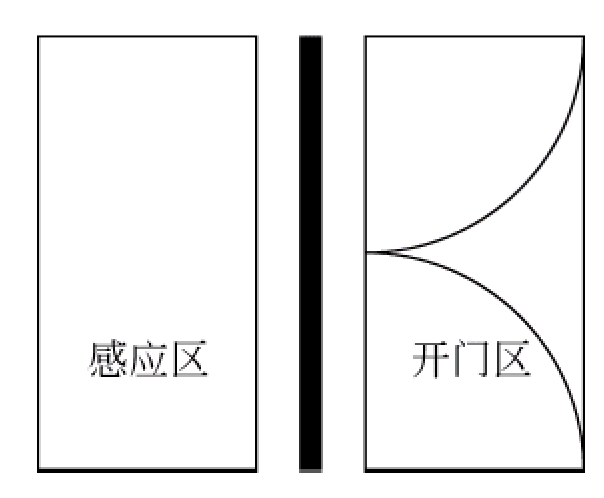


图 5-1　电子制自动门

自动感应门具有打开 O(Open)和关闭 C(Closed)两种状态,4 种可能的输入:F(Front)表示感应区内有人,R(Rear)表示开门区有人,B(Both)表示门两边均有人,N(Neither)表示门两边均没有人。现在根据上述需求设计一个用于控制该自动门的软件。

当门处于关闭 C 状态时,可能有以下几种情况。

(1) 门两边均没有人时,即系统收到 N 输入,门仍处于 C 状态。

(2) 当开门区有人时,为了防止门碰撞到人,门必须处于 C 状态。

(3) 当门两边均有人时,为了防止门碰撞到人,门也必须处于 C 状态。

(4) 当门感应区有人时,且开门区没有人,门将自动打开,转变成 O 状态。

当门处于打开 O 状态时,可能有以下几种情况。

(1) 当门感应区有人时,门将仍处于 O 状态。

(2) 当门两边均有人时,为了防止门碰撞到人,门将仍处于 O 状态。

(3) 当开门区有人时,为了防止门碰撞到人,门将仍处于 O 状态。

(4) 当门两边均没有人时,门将自动关闭,转变成 C 状态。

上述自动门的状态转移可以用表格表示,如表 5-1 所示。在表格中一行表示一个状态的转变情况,状态和输入信号的交叉点表示系统处于在状态时,接收到了输入信号而转变成的下一个状态。例如,第 1 行第 2 列 O 的含义表示:当前状态关闭 C,接收到输入信号 F(Front)信号以后,转变成打开 O(Open)状态。

表 5-1 自动门的状态转移表

现状态	输入信号/后继状态			
	N(Neither)	F(Front)	R(Rear)	B(Both)
C(Closed)	C	O	C	C
O(Open)	C	O	O	O

为了清晰描述门的状态变化,用一个比较形象的图形来表示,如图 5-2 所示。

在实际应用中,单向的门无法满足实际需求,现在对自动门进一步优化,将自动门由单向升级为双向。无论人员从里向外或者从外向里均能够自动打开门,并且也不能撞伤进出门口的人员,如图 5-3 所示。

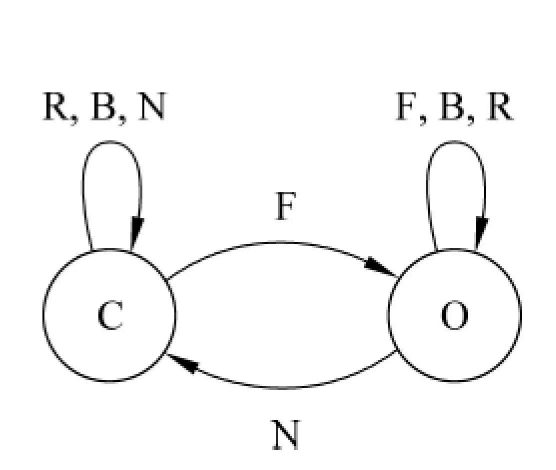


图 5-2 自动门的状态图

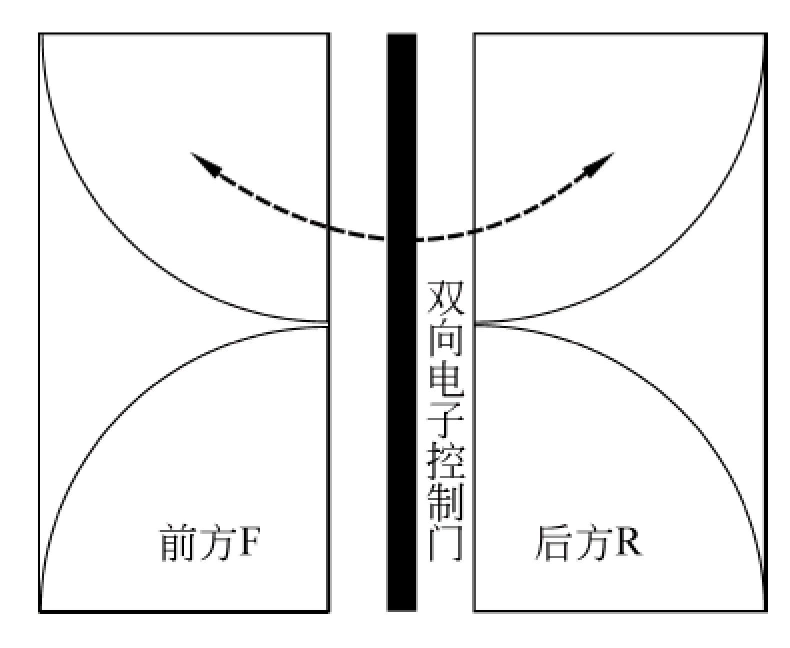


图 5-3 双向电子控制门

双向电子控制门由于门既可以向里打开,也可以向外打开。不同打开方向的控制要求是不同的,在双向电子门控制软件建模中,需要建立两个门打开状态。门向外开时的状态记为 OF(Open Front),向里面开时的状态记为 OR(Open Rear)。分析门不同的变化状态,形成了新的状态转移表,如表 5-2 所示。

表 5-2 双向电子门的状态转换表

现状态	输入信号/后继状态			
	N(Neither)	F(Front)	R(Rear)	B(Both)
C(Closed)	C	OR	OF	C
OF(Open Front)	C	OF	OF	OF
OR(Open Rear)	C	OR	OR	OR

根据这个状态转换表，可以画出双向电子控制门的有限状态机，如图5-4所示，在该状态机中门打开的状态有两个。

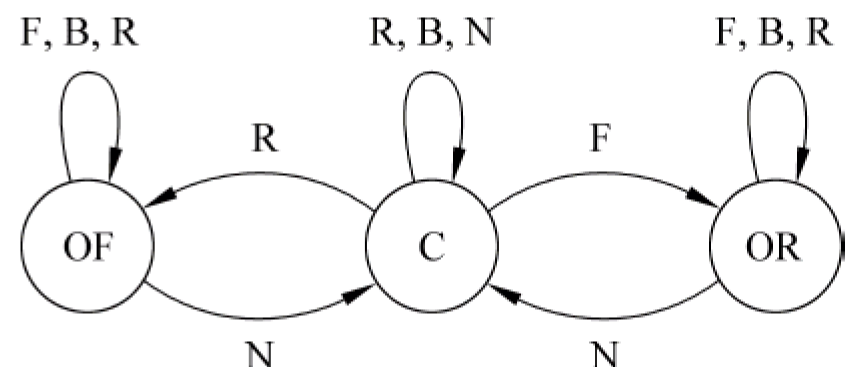


图5-4 双向电子控制门的状态转换机

一个状态机在形式上由5个部分组成：状态集、输入字母表、转移规则、初始状态和接受状态。有限状态机的状态集描述系统处于的不同状态。状态机具有两个特殊的状态：一个起始状态和一个接收状态。状态转移规则：描述系统接收不同输入信息时，从一个状态转移到另一个状态的规则。为了接收外界的输入，状态机应具备输入符号集(也称为字母表)：描述系统接收的不同输入信息。一般而言，指明了状态机允许的输入符号。

为了表示有限状态机，先给出字母表和字符串两个定义。字母表是任意一个有穷集合，字母表的成员是该字母表的符号。例如：

$$\Sigma_1 = \{0,1\}$$

$$\Sigma_2 = \{a,b,c,d,e,f\}$$

$$\Sigma_3 = \{0,1,x,y\}$$

字母表上的字符串是字母表中符号的有穷序列。例如，001100是字母表Σ_1上的一个字符串，ababaaecc是Σ_2上的一个字母串。长度为零的字符串称为空串，记为ε。空串的含义和0在数学中的含义类似。

一个有限状态机FSM是一个5元组(Σ,S,S_0,δ,F)，其中：

(1) Σ是输入字母表(符号的非空有限集合)。

(2) S是状态的非空集合。

(3) S_0是初始状态，它是S的元素。

(4) δ是状态转移函数：δ：$S\times\Sigma\rightarrow S$。

(5) $Z\subseteq S$且$Z\neq\phi$，Z是S的一个子集，是一个终态集，或叫结束集。当状态机接收完所有的输入，处于终态时，表示该输入串被状态机所接受。

输入字母表示一种输入信号，可以用简单的一个字母表示，也可以用具体包含系统特定含义的单词来表示。在自动门例子中，打开状态可以用字母O表示，也可以用单词Open表示。

如果有限状态机在字母a的输入下从状态S_1转移到状态S_2，表示状态机在状态S_1遇到了外界输入字母a，状态变更成为S_2。将其表示成$\delta(S_1,a)=S_2$。一个状态机可以用状态图表示。状态用圆圈表示，双圈表示终态集，初始状态用一个无出发点的箭头表示。状态的转移用带箭头的直线或者曲线表示，和曲线相关联的字母表示输入。

具有终态集合的FSM称为接收器和识别器(也叫作序列检测器)，它产生一个二元输出，以“是”或“否”来回答输入是否被机器接受。所有FSM的状态被称为要么接受要么不接受。在所有输入都被处理了的时候，如果当前状态是接受状态，输入被接受，否则被拒绝。

例5-1 一个有限状态机$M_1=(\Sigma,S,S_0,\delta,F)$，如图5-5所示。其中：

(1) $\Sigma=\{a,b,c,d\}$。

（2）$S=\{S_0,S_1,S_2,S_3\}$。

（3）S_0 是初始状态。

（4）$Z=\{S_3\}$是终态集。

（5）状态转移函数由表 5-3 给出。

表 5-3　M_1 的状态转移表

源状态	输入	目的状态
S_0	a	S_1
S_0	c	S_2
S_1	d	S_1
S_1	b	S_3
S_2	d	S_3

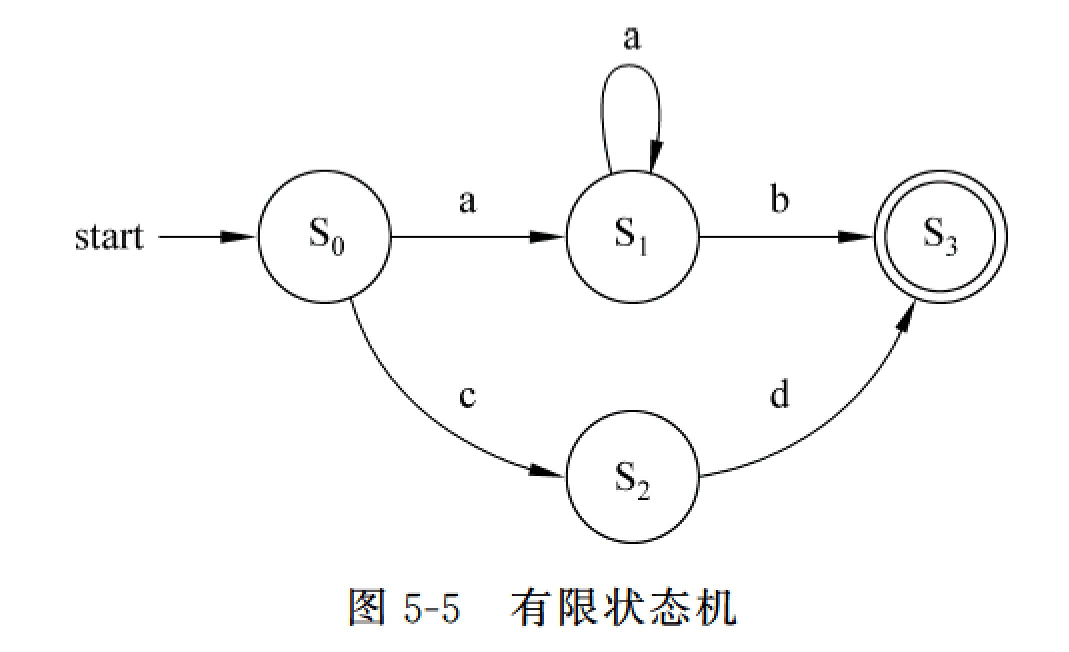


图 5-5　有限状态机

状态机 M_1，以 S_0 作为初始状态，S_3 为终止状态。它接收以下两种类型的字符串。

（1）以字符 a 开始，字符 b 结束，中间包含 0 个或者多个 d 的字符串。

（2）字符串 cd。

5.1.2　确定有限状态机和非确定有限状态机

依据状态转化过程中下一个状态是否唯一，可以将状态机分为确定有限状态机和非确定有限状态机。若一个有限状态机在每一个特定状态下，系统接收到一个输入符号（输入信号）时，下一个状态是唯一的，则称该状态机是确定有限状态机。

一个确定的有限状态机 DFSM 是一个 5 元组(Σ,S,S_0,δ,F)，其中：

（1）Σ 是输入字母表（符号的非空有限集合）。

（2）S 是状态的非空集合。

（3）S_0 是初始状态，它是 S 的元素。

（4）δ 是状态转移函数：δ：$S\times\Sigma\rightarrow S$。

（5）$Z\subseteq S$ 且 $Z\neq\phi$，Z 是 S 的一个子集，是一个终态集，或叫结束集。

给定一个确定的有限状态机 M，对于 Σ 中的字符串 t，若存在一条从初始节点到某一终止节点的路径，且这条路径上所有弧的标记符连接成的字符串等于 t，则称 t 可为 M 所接受（识别）。M 所能接受的字符串的集合记为 $L(M)$。

对于一个确定有限状态机，具有以下几个特征。

（1）初始状态唯一。

（2）确定有限状态机的特征，对任何状态 $S_i\in S$ 和输入符号 a，$\delta(S_i,a)$唯一地确定了下一个状态，即转换函数至多确定一个状态。

（3）没有空边，即没有输入 ε。

相对于确定性有限状态机，非确定有限状态机对于一个特定的状态的一个输入字母表，状态可能迁移到多个状态中的任意一个。如图 5-6 所示的有限状态机 M 处于状态 S_1

下，接收了输入 a 以后可以转换到状态 S_2 和状态 S_3，从 S_1 将会转移到哪个状态是不确定的。非确定有限状态机是 Michael O. Rabin 和 Dana Scott 在 1959 年引入的。

图 5-6 S_1 输入 a 会有两种后继状态

这种性质可以用幂集表示。在数学上，给定集合 S，其幂集$\mathbb{P}(S)$是以 S 的全部子集为元素的集合。以符号表示即为：

$$\mathbb{P}(S)=\{U \mid U\subseteq S\}$$

例如，S 是集合$\{S_1, S_2, S_3\}$，则$\mathbb{P}(S)=\{\varnothing, \{S_1\}, \{S_2\}, \{S_3\}, \{S_1, S_2\}, \{S_1, S_3\}, \{S_2, S_3\}, \{S_1, S_2, S_3\}\}$。其中，$\varnothing$表示空集。在图 5-6 的例子中，可以表示为 $\delta(S_1, a)=\{S_2, S_3\}$。因此在非确定有限状态机中，转移函数为多值函数。

NFSM 允许到新状态的变换不消耗任何输入符号。例如，如果它处于状态 S_1，下一个输入符号是 a，它可以移动到状态 S_2 而不消耗任何输入符号。不消耗输入符号的到新状态的变换叫作 ε 转移或 λ 转移。在如图 5-7 所示的 NFSM 中，S_2 接受输入 0 将会变化到状态 S_3，在不接受输入时（或者说接受空串）也可以变化到状态 S_3，换言之，状态可以由 S_2 以多种方法转换到 S_3。

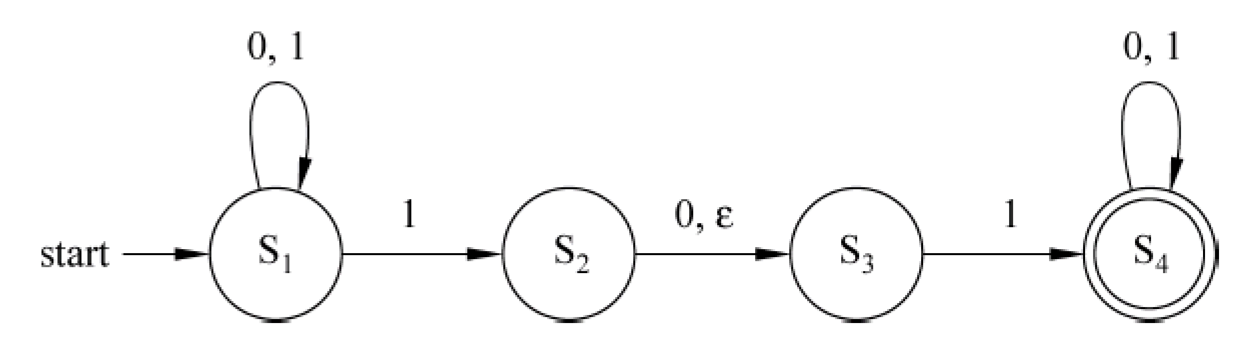


图 5-7 带有空串的 NFSM

一个非确定的有限状态机是一个 5 元组$(\Sigma, S, S_0, \delta, F)$，其中：

(1) Σ 是包含的输入字母表。

(2) S 是状态的非空集合。

(3) δ 是状态转移函数 δ：$S\times(\Sigma\cup\{\varepsilon\})\rightarrow\mathbb{P}(S)$。$\delta$ 是一个多值函数。

(4) $Z\subseteq S$ 且 $Z\neq\phi$，Z 是 S 的一个子集，是一个终态集，或叫结束集。

对于图 5-7 的 NFSM，其形式描述是$(\Sigma, S, S_0, \delta, F)$，其中：

(1) $\Sigma=\{0, 1\}$。

(2) $S=\{S_1, S_2, S_3, S_4\}$。

(3) δ 由表 5-4 给出。

表 5-4 NFSM 的状态转换表

状态	输入		
	0	1	ε
S_1	$\{S_1\}$	$\{S_1, S_2\}$	ϕ
S_2	$\{S_3\}$	ϕ	$\{S_3\}$
S_3	ϕ	$\{S_4\}$	ϕ
S_4	$\{S_4\}$	$\{S_4\}$	ϕ

5.1.3 确定有限状态机和非确定有限状态机的转换

设 M_1 和 M_2 是同一个字母表上的自动机，如果有 $L(M_1)=L(M_2)$，则称 M_1 和 M_2 等价。尽管 DFSM 和 NFSM 有不同的定义，在形式理论中可以证明它们是等价的。Michael O. Rabin 和 Dana Scott 已经证明了它与确定自动机的等价性。显然 DFSM 是 NFSM 的一种特例，任何 DFSM 都是一个 NFSM，比较容易理解。对于任何给定 NFSM，都可以构造一个等价的 DFSM，从表面上看似乎不成立，似乎 NFSM 的表达能力强于 DFSM。对于 NFSM、DFSM 两者的等价性证明读者可以参考相关文献，本书仅介绍从 NFSM 到 DFSM 的转换算法。

情况 1：假设 NFSM 不包含 ε 转移。对于一个非确定的有限状态机 NFSM $M_1=(\Sigma, S, S_0, \delta, F)$，构造一个和 M_1 等价的确定有限状态机 $M_2=(\Sigma', S', S_0', \delta', F')$。

其构造算法如下。

(1) M_2 的输入字母表和 M_1 相同，$\Sigma'=\Sigma$。

(2) M_2 的初始状态由 M_1 的初始状态的集合构成，即 $S_0'=[S_0]$。

(3) M_2 的迁移函数 δ'：对于一个输入字母，若 $\delta(\{S_1, S_2, \cdots, S_i\}, a)=\{R_1, R_2, \cdots, R_j\}$，定义 $\delta'=([S_1, S_2, \cdots, S_i], a)=[R_1, R_2, \cdots, R_j]$。如果简记 $q_i=[S_1, S_2, \cdots, S_i]$，$q_j=[R_1, R_2, \cdots, R_j]$，可以表达成 $\delta'(q_i, a)=q_j$。

(4) M_2 的状态由 M_1 的状态的幂集构成。从 S_0' 出发，根据迁移函数 δ' 所遍历的过程，构成 M_2 的状态集合。

(5) M_2 的终止集合，若 M_2 中的某一个状态 $[R_1, R_2, \cdots, R_j]$ 包含 M_1 的终止状态，则 $[R_1, R_2, \cdots, R_j]\in F'$。也就是 $F'=\{[R_1, R_2, \cdots, R_j] \mid [R_1, R_2, \cdots, R_j]\in S'$ 且 $\{R_1, R_2, \cdots, R_j\}\cap F\neq\varnothing\}$。

例 5-2 $M_1=(\{a, b\}, \{S_1, S_2, S_3\}, S_1, \delta, \{S_2\})$，其中 $\delta(\{S=\}, a)=\{S_1, S_2\}$，如图 5-8 所示。现将其转换成等价的 DFSM。

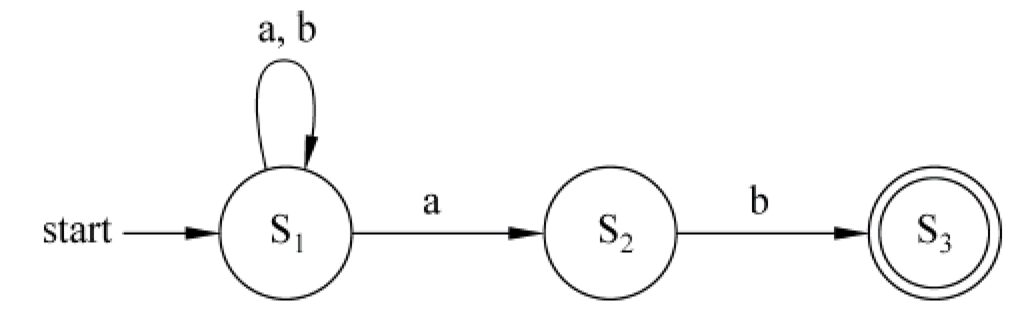


图 5-8 要转换的 NFSM

转换步骤如下。

(1) M_2 的输入字母表和 M_1 相同，$\Sigma'=\{a, b\}$。

(2) M_1 状态的幂集为 $S'=\{q_1=[S_1], q_2=[S_2], q_3=[S_3], q_4=[S_1, S_2], q_5=[S_1, S_3], q_6=[S_2, S_3], q_7=[S_1, S_2, S_3], \varnothing\}$。

(3) M_2 的初始状态为 $q_1=[S_1]$。

构造状态转移函数 δ'，从 q_1 开始，根据其输入字母表，不断得到新的状态。例如，q_1 输入 a 得到 q_4 状态，输入 b 得到 q_1 状态，如图 5-9(a)所示。q_1 由于是其本身，不用考虑，

而 q_4 在输入 a 和 b 后分别得到 q_4 和 q_5，如图 5-9(b)所示。

	a	b
q_1=[S_1]	q_4	q_1
q_2=[S_2]		
q_3=[S_3]		
q_4=[S_1, S_2]		
q_5=[S_1, S_3]		
q_6=[S_2, S_3]		
q_7=[S_1, S_2, S_3]		
∅		

(a)

⇨

	a	b
q_1=[S_1]	q_4	q_1
q_2=[S_2]		
q_3=[S_3]		
q_4=[S_1, S_2]	q_4	q_5
q_5=[S_1, S_3]		
q_6=[S_2, S_3]		
q_7=[S_1, S_2, S_3]		
∅		

(b)

图 5-9 状态转换过程 1

同理，q_4 已经遍历过了，只要考虑 q_5 就可以了。而 q_5 在输入的作用下，分别得到 q_4 和 q_1。由于从 q_1 所有的边都已经遍历过了，构造完成。而 q_2，q_3，q_6，q_7 由于无法到达而丢弃，如图 5-10 所示。

	a	b
q_1=[S_1]	q_4	q_1
q_2=[S_2]		
q_3=[S_3]		
q_4=[S_1, S_2]	q_4	q_5
q_5=[S_1, S_3]		
q_6=[S_2, S_3]		
q_7=[S_1, S_2, S_3]		
∅		

(a)

⇨

	a	b
q_1=[S_1]	q_4	q_1
q_2=[S_2]		
q_3=[S_3]		
q_4=[S_1, S_2]	q_4	q_5
q_5=[S_1, S_3]	q_4	q_1
q_6=[S_2, S_3]		
q_7=[S_1, S_2, S_3]		
∅		

(b)

图 5-10 状态转换过程 2

最后形成的确定有限状态机 $M=(\{a,b\},\{q_1,q_2,q_3\},q_1,\delta,\{q_5\})$，如图 5-11 所示。

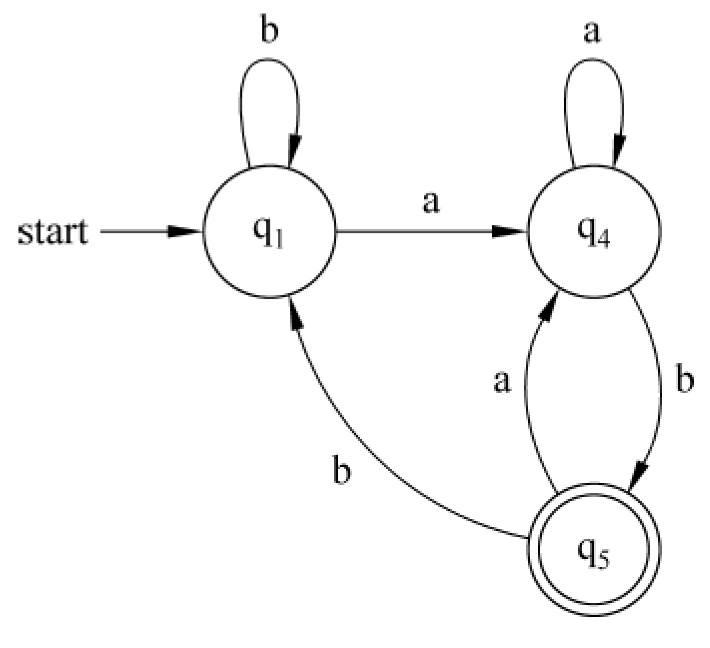


图 5-11 转换以后的 DFSM

情况 2：假设 NFSM 包含 ε 转移。对于一个非确定的有限状态机 NFSM $M_1=(\Sigma,S,S_0,\delta,F)$，构造一个和 M_1 等价的确定有限状态机 $M_2=(\Sigma',S',S_0',\delta',F')$。

其构造思路如下：首先把从 S_0 出发，仅经过任意条 ε 边所能到达的状态所组成的集合作为 M_2 的初态 S_0'，然后分别把从 S_0' 出发，经过对输入符号 $a\in\Sigma$ 的状态转移所能到达的状态（包括转移后再经 ε 边所能到达的状态）组成的集合作为 M_2 的状态，如此递归，直到不再有新的状态出现为止。

为了方便描述转换过程，先定义 ε-闭包。

状态 S_i 的 ε-闭包：从一个状态 S_i 出发，经过全部标记 ε 的边所能够到达的所有状态的集合(包括 S_i 本身)，记为 ε-Closure(S_i)。形式上，对于状态 S_i，定义：

(1) 对于状态 S_i：$S_i \in$ ε-Closure(S_i)。

(2) 如果 $\forall S_i \in$ ε-Closure(S_i)，且 $\delta(S_i, \varepsilon) = S_j$，则 $S_j \in$ ε-Closure(S_i)。

对于状态集合 S，其 ε-闭包定义为：

$$\varepsilon\text{-Closure}(S) = \bigcup_{S_i \in S} \varepsilon\text{-closure}(S_i)$$

构造算法如下。

(1) M_2 的输入字母表和 M_1 相同，$\Sigma' = \Sigma$。

(2) M_2 的初始状态由 M_1 的初始状态的 ε-闭包构成，即 $S_0' =$ ε-Closure(S_0)。

(3) M_2 的迁移函数 δ'：对于状态 S_i' 和 ε，$\delta'(S_i, \varepsilon) =$ ε-Closure(S_i)，对于一个状态 S_i 输入字母 a，$\delta'(S_i', a) =$ ε-Closure(P)，其中 $P = \delta(S_i', a)$。

(4) M_2 的状态 S' 由 M_1 的状态的幂集构成。从 S_0' 出发，根据迁移函数 δ' 所遍历的过程，构成 M_2 的状态集合。

(5) M_2 的终止集合等定义，若 M_2 中的某一个状态$[R_1, R_2, \cdots, R_j]$包含 M_1 的终止状态，则$[R_1, R_2, \cdots, R_j] \in F'$。也就是 $F' = \{[R_1, R_2, \cdots, R_j] \mid [R_1, R_2, \cdots, R_j] \in S'$ 且 $\{R_1, R_2, \cdots, R_j\} \cap F \neq \varnothing\}$。

其中第(4)步，可以详细描述如下。

(1) $S_0' =$ ε-Closure(S_0)。

(2) 将 S_0' 添加到 S'。

(3) 对于 S' 中未访问的状态 $S_i' = \{S_{i1}, S_{i2}, \cdots, S_{im}\}$，做访问标记，并对于每一个输入字母 $a \in \Sigma'$，$P = \delta'(S_i', a) = \delta'(\{S_{i1}, S_{i2}, \cdots, S_{im}\}, a)$，$q_i =$ ε-Closure(P)，若 q_i 不在 S' 中，将 q_i 作为未做标记的状态添加到 S' 中。在该步骤中，采用深度优先或者广度优先均可。

(4) 重复步骤(3)，直到 S' 不再包含未标记的状态。

例 5-3 现有 NFSM，$M_1 = (\Sigma, S, S_0, \delta, F) = (\{a, b, \varepsilon\}, \{S_1, S_2, S_3, S_4\}, S_1, \delta, \{S_3\})$，δ 定义如表 5-5 所示，带有 ε 的 NFSM 如图 5-12 所示。

表 5-5 NFSM 的状态转换表

状态	输入			
	a	b	c	ε
S_1	$\{S_1\}$	ϕ	ϕ	$\{S_2, S_4\}$
ϕ		$\{S_2, S_3\}$	ϕ	ϕ
S_3	ϕ	ϕ	ϕ	ϕ
S_4	ϕ	ϕ	$\{S_4\}$	$\{S_3\}$

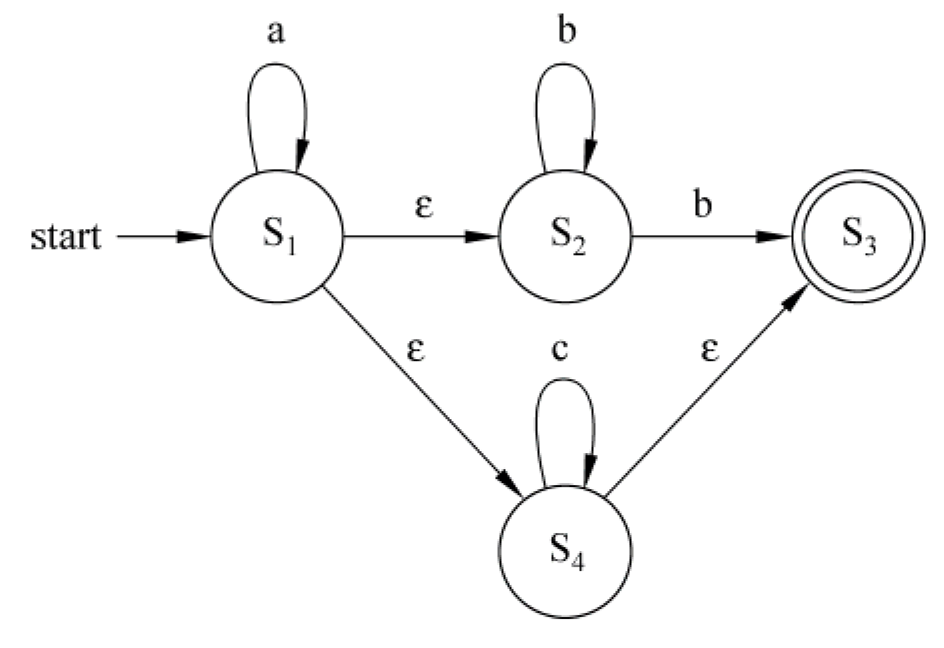


图 5-12 带有 ε 的 NFSM

现在构造与其等价的 DFSM，$M_2 = (\Sigma', S', S_0', \delta', F')$。

(1) $S_0' =$ ε-Closure(S_0) $= \{S_1, S_2, S_3, S_4\} = q_1$。

(2) $\Sigma' = \{a, b, c\}$。

(3) $S'=\{S_0'\}=\{q_1\}$。

(4) 标记 q_1,取出 q_1,并计算各个迁移的 $\delta'(q_1,a)=q_1$,$\delta'(q_1,b)=q_2=\{s_1,s_2\}$,$\delta'(q_1,c)=q_3=\{s_2,s_3\}$。并将 q_1 和 q_3 添加到 S' 中,$S'=\{q_1, q_2, q_3\}$,已经访问的集合 marked$=\{q_1\}$。

(5) 标记 q_2,取出 q_2,并计算各个迁移的 $\delta'(q_2,a)=q_1$,$\delta'(q_2,b)=\phi$,$\delta'(q_2,c)=\phi$。由于 q_2 已经在 S' 中,$S'=\{q_1, q_2, q_3\}$,S' 保持不变。已经访问的集合 marked$=\{q_1, q_2\}$。

(6) 标记 q_3,取出 q_3,并计算各个迁移的 $\delta'(q_3,a)=\phi$,$\delta'(q_3,b)=\phi$,$\delta'(q_3,c)=\phi$。由于 q_3 已经在 S' 中,$S'=\{q_1, q_2, q_3\}$,S' 保持不变。已经访问的集合 marked$=\{q_1, q_2, q_3\}$。S' 中的所有元素均已经访问完毕,过程如图 5-13 所示。

(7) NFSM 中的终止集为$\{S_1\}$,而 q_1, q_2, q_3 均包含 M_1 的终态集$\{S_3\}$,所有 q_1, q_2, q_3 均为 M_2 的终态集。最后构造的 M_2 如图 5-14 所示。

	a	b	c
$q_1=[S_1, S_2, S_3, S_4]$	q_1	q_2	q_3
$q_2=[S_1, S_2]$			
$q_3=[S_2, S_3]$			

	a	b	c
$q_1=[S_1, S_2, S_3, S_4]$	q_1	q_2	q_2
$q_2=[S_1, S_2]$	$\varnothing$	q_2	$\varnothing$
$q_3=[S_2, S_3]$			

	a	b	c
$q_1=[S_1, S_2, S_3, S_4]$	q_1	q_2	q_2
$q_2=[S_1, S_2]$	$\varnothing$	q_2	$\varnothing$
$q_3=[S_2, S_3]$	$\varnothing$	$\varnothing$	q_3

图 5-13　状态转换过程 1

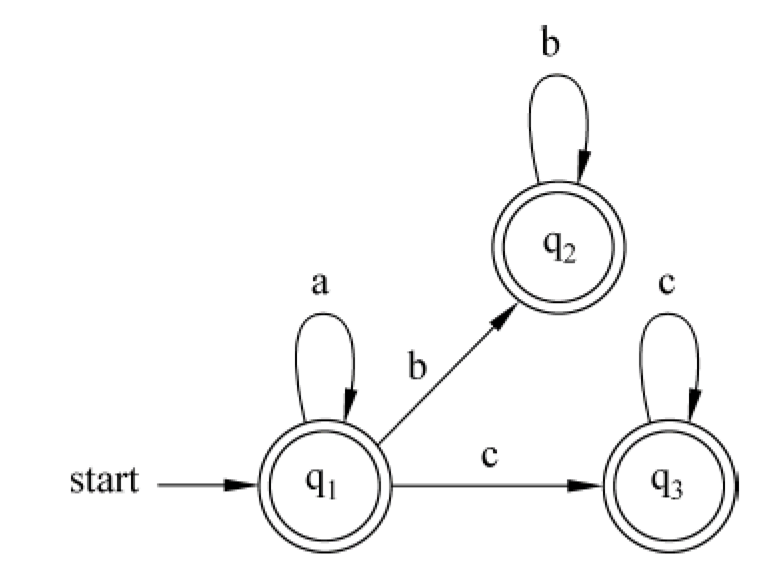


图 5-14　转换以后的 DFSM

5.1.4　带状态输出的有限自动机

现实生活中的许多有限状态系统,对于不同的输入信息,除内部状态不断改变外,还不断向系统外部输出各种信息。根据输出是否和输入信号有关,确定的有限状态机可以划分为摩尔型 Moore 状态机和米利型 Mealy 状态机。

1. Moore 状态机

Moore 状态机的输出仅由当前状态确定,和状态机的输入无关,其原理如图 5-15 所示。

Moore 状态机是一个 6 元组$(\Sigma,\Lambda, S,S_0,\delta, \omega)$,其中:

(1) Σ:输入字母表(符号的非空有限集合)。

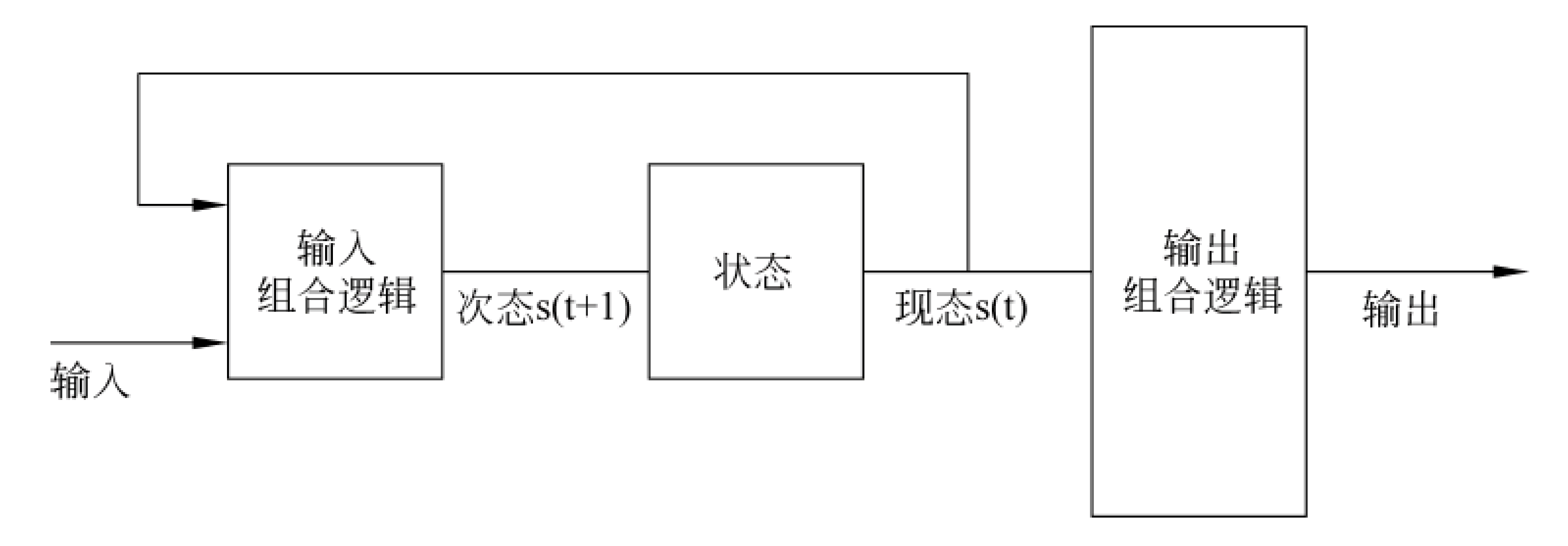


图 5-15　Moore 状态机原理

(2) Λ：输出字母表的有限集合。

(3) S：状态的非空集合。

(4) S_0：初始状态，它是 S 的元素。

(5) δ：状态转移函数，δ：S×Σ→S。

(6) ω：输出函数，映射每个状态到输出字母表，(ω：S →Λ)。

例 5-4　设有一个 Moore 状态机，其中：

(1) 输入字母表 Σ={x,y,z}。

(2) 输出字母表 Λ={a,b,c}。

(3) 状态集合：S={S_0,S_1,S_2,S_3}。

(4) 初始状态：S_0。

(5) 状态转移函数和输出函数如表 5-6 所示。该 Moore 状态机可以用图 5-16 表示。

表 5-6　Moore 的状态转换表

状态	输入/下一状态			输出
	x	y	z	
S_0	S_3	S_3	S_1	b
S_1	S_3	S_2	S_3	a
S_2	--	--	S_3	a
S_3	--	S_2	--	c

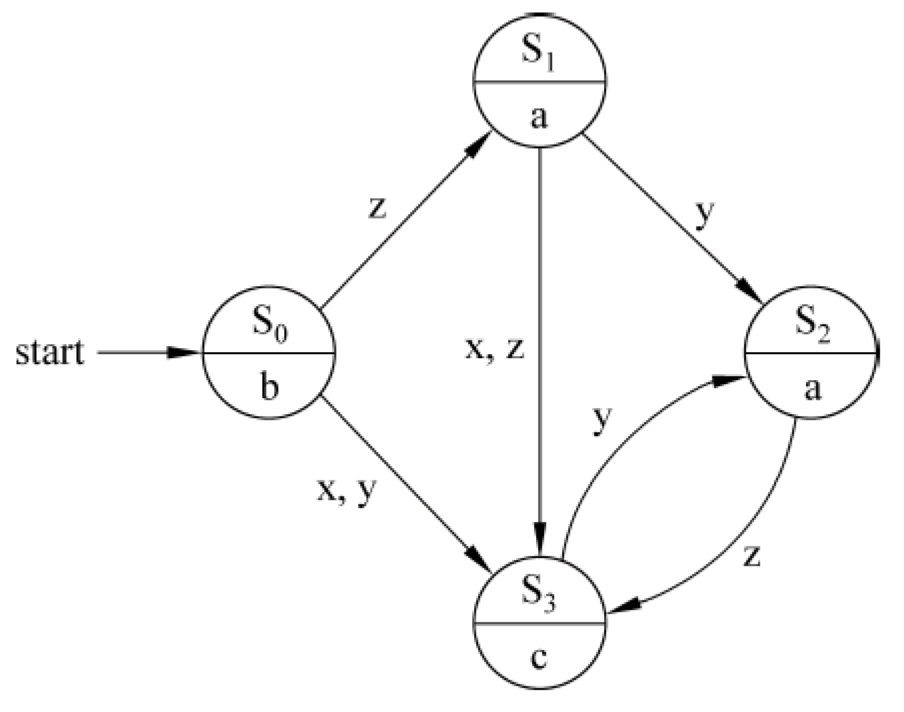


图 5-16　Moore 状态机

2. Mealy 状态机

和 Moore 状态机不同，Mealy 状态机由当前的状态和当前的输入共同决定，如图 5-17 所示。

Mealy 状态机是一个 6 元组(Σ,Λ, S,S_0,δ, ω)，其中：

(1) Σ：输入字母表(符号的非空有限集合)。

(2) Λ：输出字母表的有限集合。

(3) S：状态的非空集合。

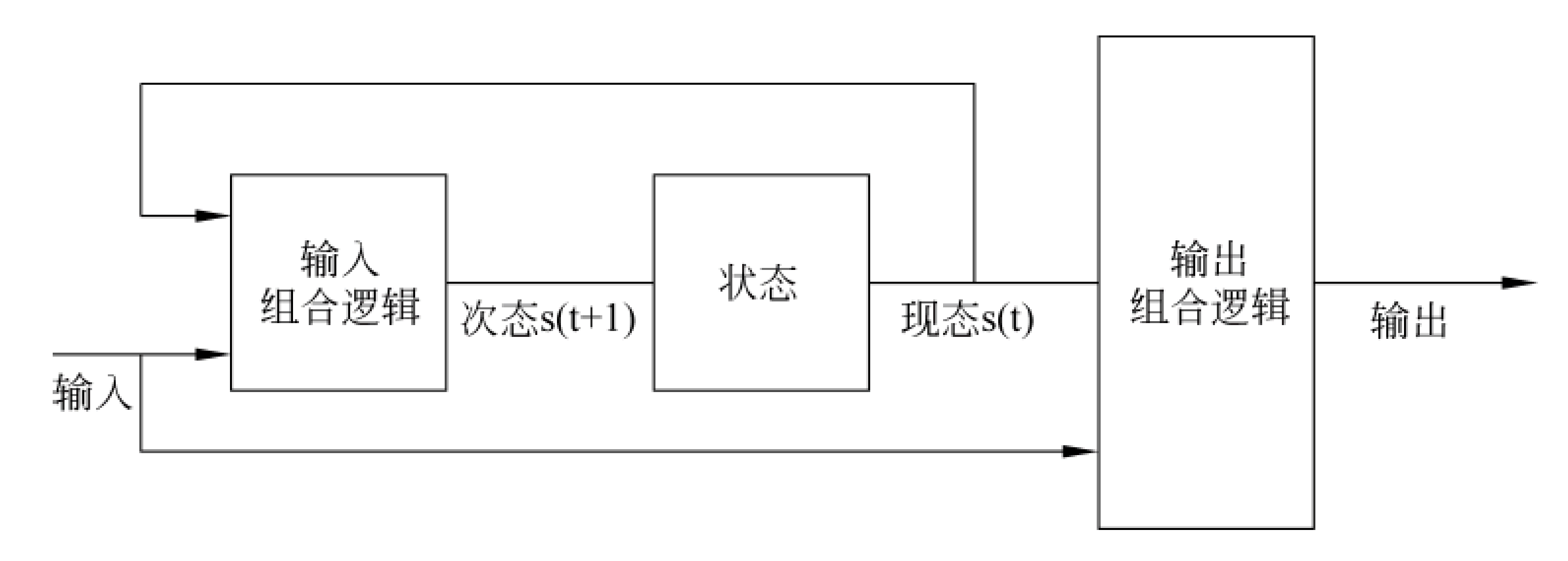


图 5-17 Mealy 状态机原理

(4) S_0：初始状态，它是 S 的元素。

(5) δ：状态转移函数，δ：S×Σ→S。

(6) ω：输出函数，映射每个状态到输出字母表，(ω：S×Σ→Λ)。

例 5-5 有一个 Mealy 状态机 M，可以描述如下。

(1) 输入字母表 Σ＝{1,0}。

(2) 输出字母表 Λ＝{1,0}。

(3) 状态集合：S＝{s_0,s_1,s_2}。

(4) 初始状态：S_0＝ s_0。

(5) 其中，转移函数和输出函数，可以表示为表 5-7 和图 5-18。

表 5-7 Mealy 状态机转移和输出

状态	输入/下一状态		输入/输出	
	1	0	1	0
s_0	s_0	s_1	1	1
s_1	s_0	s_1	0	0
s_2	s_0	s_1	0	0

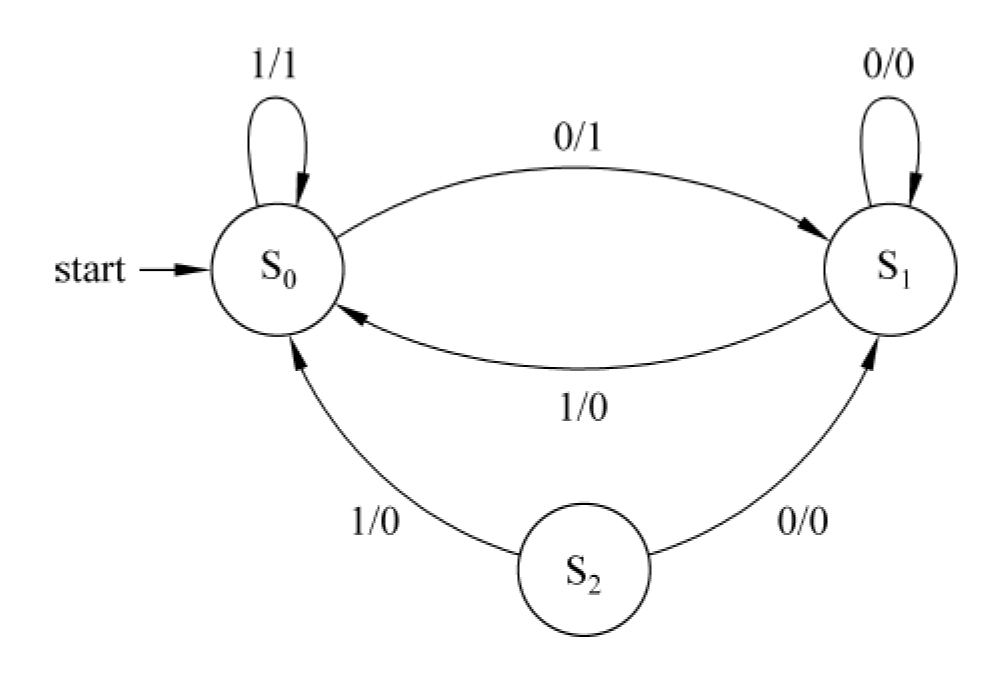


图 5-18 Mealy 状态机的例子

3. Moore 状态机和 Mealy 状态机的转换

Moore 状态机和 Mealy 状态机在表达能力上是相同的，并且它们可以相互转化。Mealy 状态机转换为 Moore 状态机，其方法如下。

(1) 若一个状态的输出相同，将该输出移到状态内部。

(2) 若一个状态存在不同的输出，将该状态分解成具有相同输出的子状态，然后将其输出移到其子状态内部。

图 5-19(a)表示的是一个 Mealy 状态机的局部，讨论状态 q_d 的变化，q_d 状态的后继状态有三个，在输入 0,2,1 的作用下分别到达 q_a,q_b,q_e，其输出都是 1，在这种情况下，可以将转移弧线上的输出，转移到状态 q_d 上，表示 q_d 状态的后继输出均为 1，而其他节点和讨论的问题无关，暂时保留为空白，形成对应的 Moore 状态图，如图 5-19(b)所示。

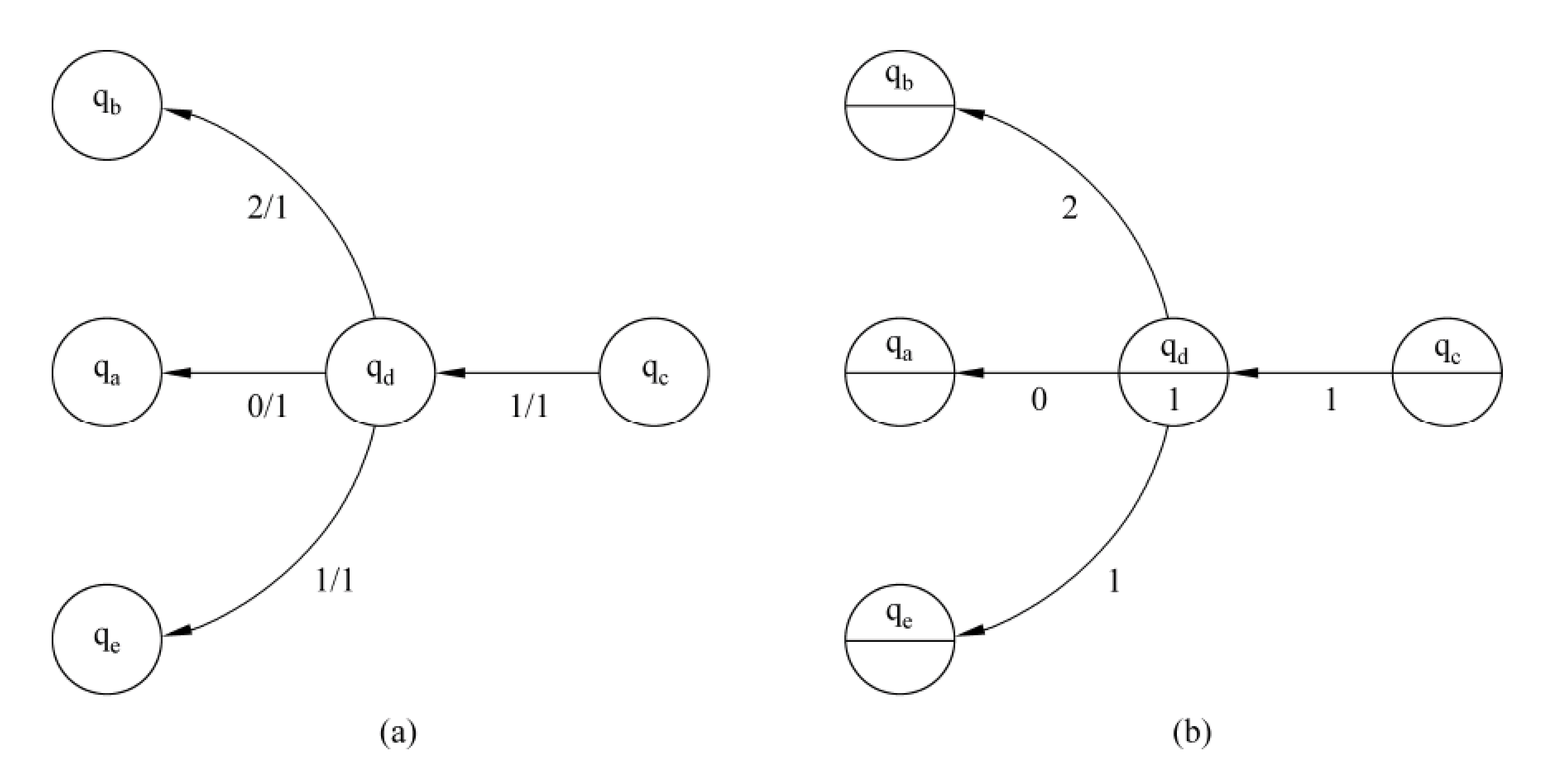


图 5-19 具有相同输出的状态转换示意图

图 5-20(a)表示的是一个 Mealy 图的局部，讨论状态 q_d 的变化。q_d 状态的后继状态有三个，在输入 0,2,1 的作用下分别到达 q_a,q_b,q_e,q_a 和 q_b 的输出是 1，而 q_e 的输出时 0。在这种情况下，可以将转移到 q_a 和 q_b 的状态作为一个子状态 q_d，其输出为 1，而输出为 0 作为另一个子状态 $q_d{}'$，形成了对应的 Moore 状态图。显然，对于其前驱节点 q_c 而言，这构成了一个非确定的状态机。前面已经讨论过了确定状态机和非确定状态机的等价性。

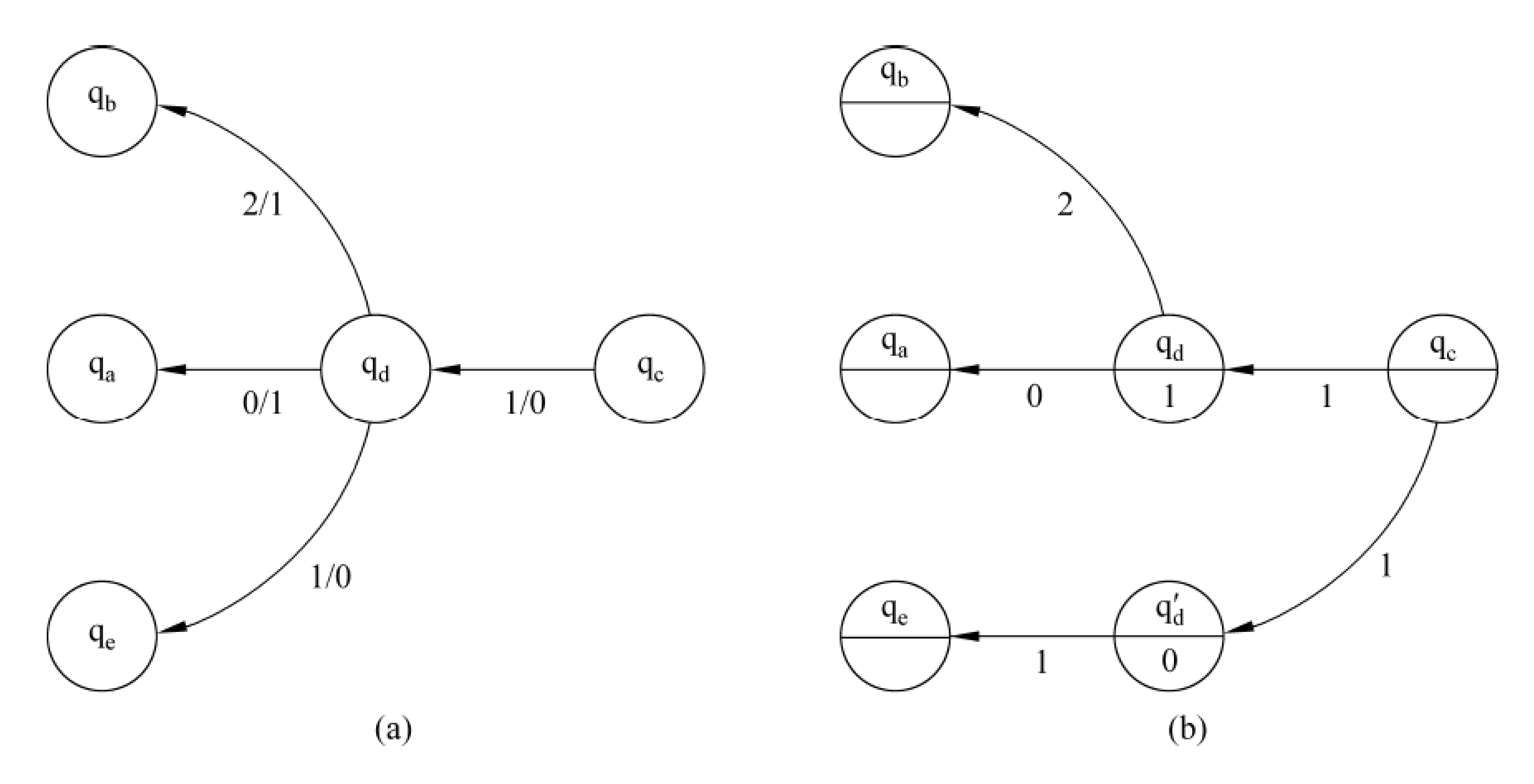


图 5-20 带有不同输出时转换

Moore 状态机转换为 Mealy 状态机，只需将输出由状态内部移到弧上即可。

例 5-6 状态机 M 是一个 Mealy 状态机，其中：

(1) 输入字母表 $\Sigma=\{1,0\}$。

(2) 输出字母表 $\Lambda=\{1,0\}$。

(3) 状态集合：$S=\{q_a,q_b,q_c,q_d\}$。

(4) 初始状态：$S_0=q_a$。

该状态转移函数和输出函数如表 5-8 所示。

状态转换可以用图 5-21 表示，状态 q_a 有 4 个箭头指向它，表示在当前状态下有 4 个状态可以转换到状态 q_a；同时当前输出均为 0，可以把 0 移入状态 q_a 内部，表示在 Moore 机中状态 q_a 的输出为 0。同理，可以把 0 分别移到 q_b 和 q_c 内部。但对于状态 q_d，有两个箭头指向且具有不同的输出值，需要把状态 q_d 分解成两个状态 q_d 和 q_e，q_d 的输出为 0，而 q_e 的输出为 1，得到完整的 Moore 机状态模型，如图 5-21 所示。

表 5-8 Mealy 的状态转换表

状态	输入/下一状态		输入/输出	
	1	0	1	0
q_a	q_a	q_b	0	0
q_b	q_c	q_a	0	0
q_c	q_d	q_a	0	0
q_d	q_d	q_a	1	0

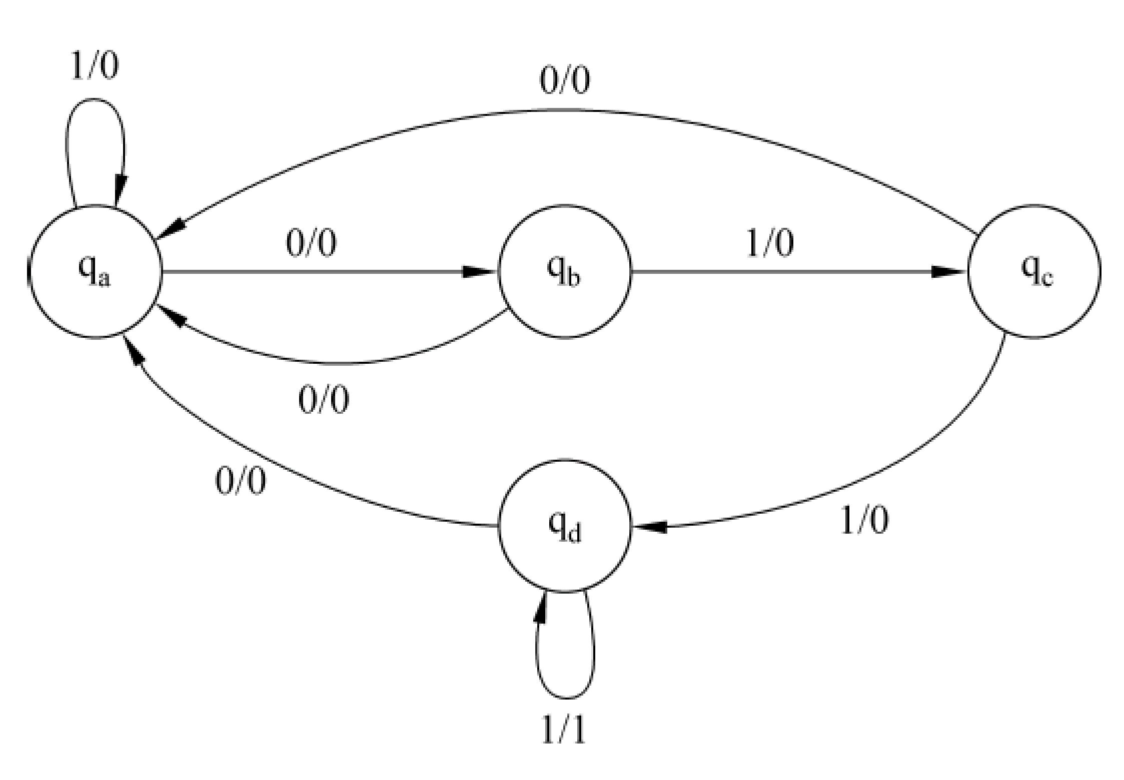


图 5-21 Mealy 状态机

在图 5-21 中描述的 Mealy 状态机转换成 Moore 状态机，如图 5-22 所示。

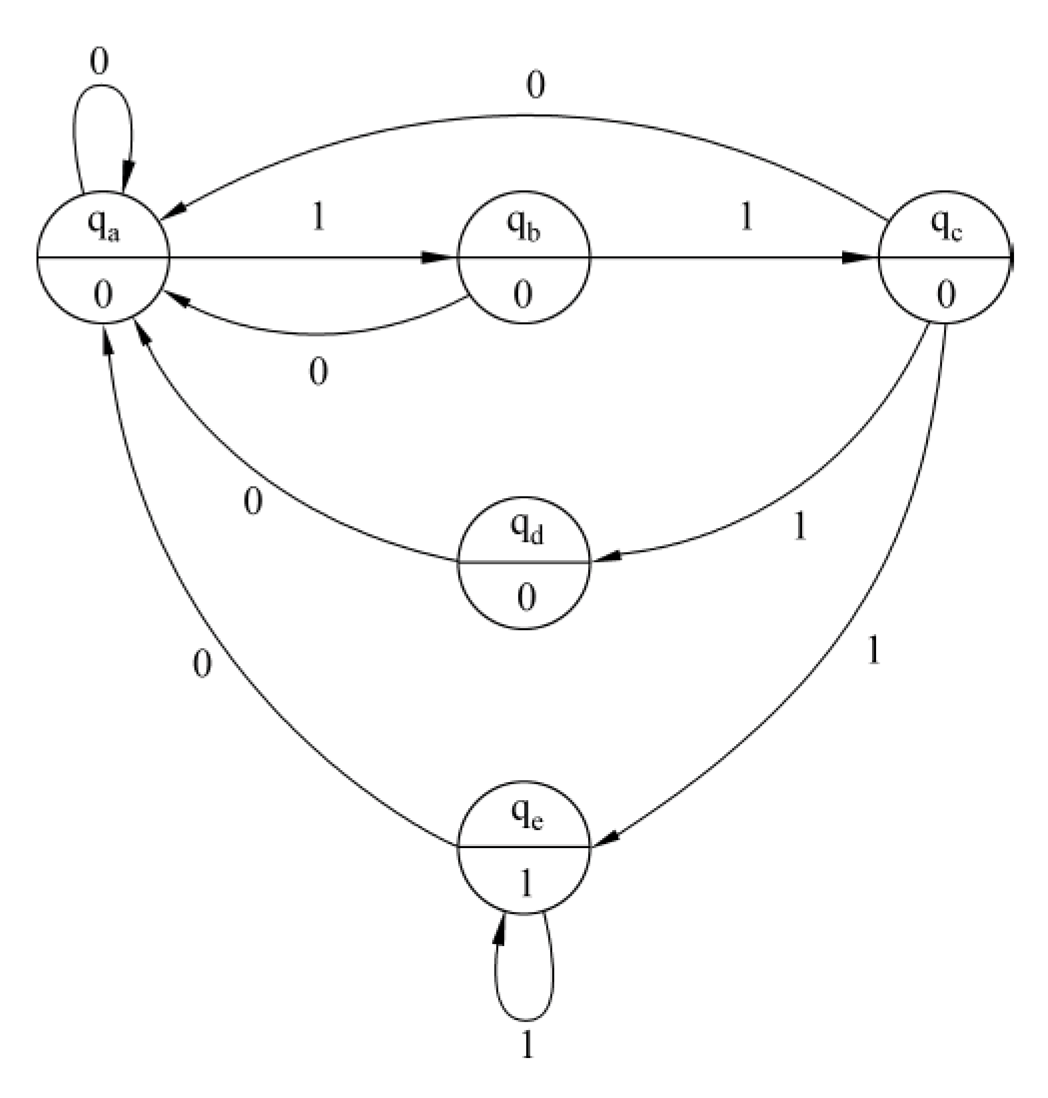


图 5-22 转变以后的 Moore 状态机

由于确定的状态机和非确定状态机，以及 Moore 状态机和 Mealy 状态机的等价性，在后继的讨论和分析过程中，均采用确定的 Mealy 状态机来描述。

5.2　基于有限状态机测试的假设和特性

基于有限状态机的测试，其本质是确认一个状态机的实现 M_I 和状态机的规格说明 M_S 之间的一致性。根据状态机的规格说明，分析状态机可能存在的故障模型，依据故障模型产生测试用例。通过执行测试用例中的输入序列，在状态机的实现和规格说明之间比较产生的后继状态及其输出一致性，从而发现可能存在的错误。

如果一个 M_I 和 M_S 等价，那么 M_I 和 M_S 必须满足以下两个条件。

（1）具有相同的状态。

（2）从对应的状态开始，对于同一个输入序列都具有相同的输出序列，并转移到对应的后继状态。

对于一个状态机而言，输入序列的数量是无限的，在实际测试过程中，只能选择有限的输入序列。基于状态机测试的关键步骤是确定有限的输入序列，输入序列产生的通常是根据 FSM 的故障模型。根据 FSM 规格说明，分析一种故障模型的集合，根据这个故障模型的集合设计出测试用例，从而判别 M_I 和 M_S 的一致性，如图 5-23 所示。

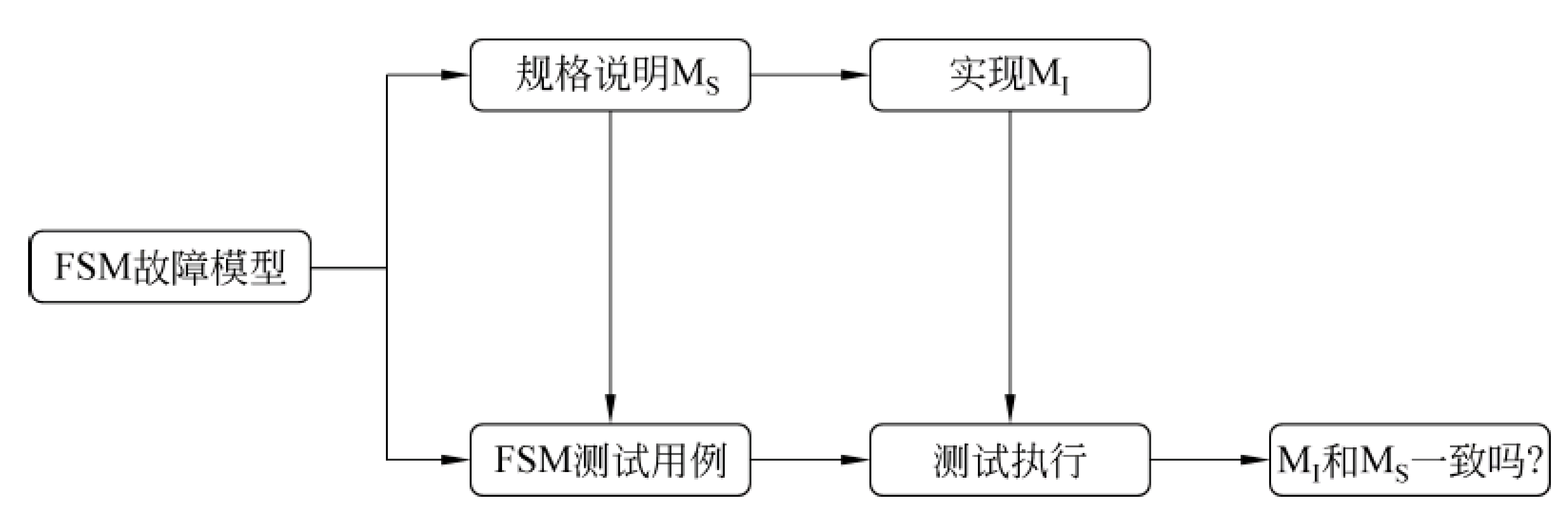


图 5-23　FSM 测试的基本原理

1. 假设 5-1 初始状态假设

初始状态：M_S 和 M_I 都有一个初始状态 S_0，且测试之前 M_I 和 M_S 已经处在它的初始状态。在本书讨论的例子中，除非特殊说明，编号（包括字母编号）最小的状态为初始状态。

2. 假设 5-2 可重置性假设

可重置性假设：M_S 和 M_I 都有一个重置输入 reset（简称 r），使得状态机在任何状态时都能回到初始状态 S_0 且不会产生任何输出。

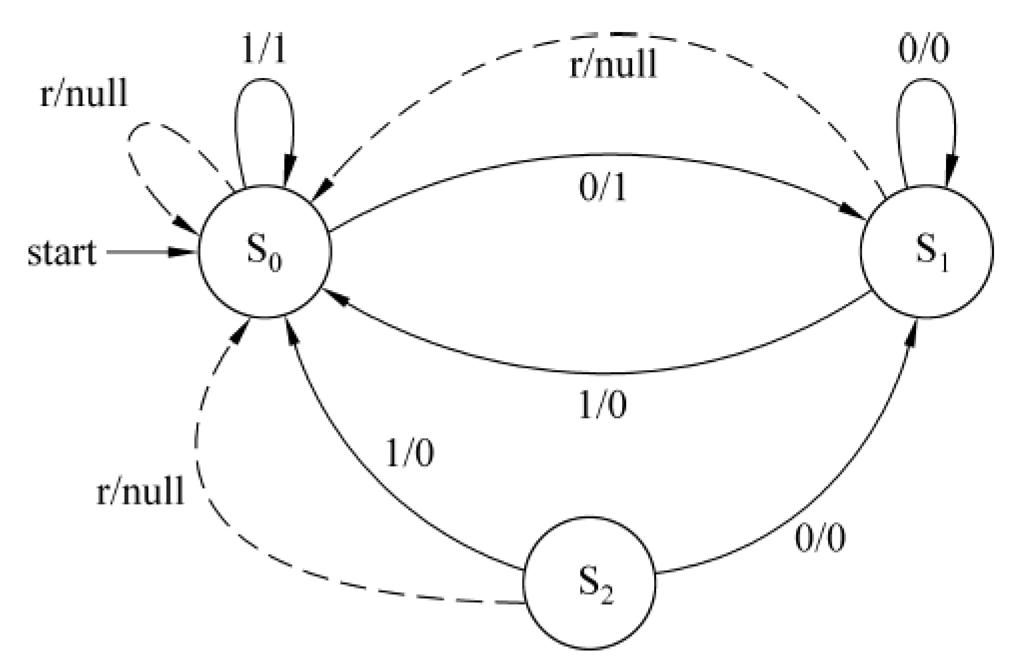


图 5-24　可重置的有限状态机

图 5-24 表示的就是一个带有重置输入的有限状态机，重置转换在图中用虚线表示，r 表示重置输入 reset，null 表示没有任何

输出。在任何状态下,均可以通过重置输入使得状态机的状态回到起始状态。但在后面的讨论过程中,为了清晰表示状态机,一般情况下都不画出重置转换。

3. 假设 5-3 完备性假设

给定一个有限状态机 M_S,满足:

$$\forall s_i \in S:(\forall a \in \Sigma:(\exists s_j \in S:(\delta(s_i,a)=s_j)))$$

则称有限状态机 M_S 是完备的。M_S 是完备的,对于 M_S 的每一个状态,都存在和不同输入相对应的迁移。

在后面的测试讨论中假设有限状态机是完备的。

4. 假设 5-4 精简(最小化)假设

给定一个有限状态机 M_S,满足:

$$\forall s_i,s_j \in S:(\exists a \in \Sigma:(\delta(s_i,a)\neq\delta(s_j,a)))$$

则称有限状态机 M_S 是精简的。即在 M_S 中,任意两个状态,都存在一个输入,使得 M_S 的输出不同。换言之,不存在两个或者以上的节点是等价的。

5.3 有限状态机的故障模型

给定一个有限自动机 M_S,软件开发人员根据 M_S 实现了一个具体的系统,这个系统从本质上而言,也是一个有限自动机,标记为 M_I。在某些时候,M_I 就是 M_S 的完全模拟,M_I 和 M_S 在输出和状态转换上完全一样,而在有些时候,M_I 和 M_S 存在比较大的差异,或者 M_I 在实现时存在不同类型的错误。在本章中所讨论的问题,假设 M_S 的设计是正确的。

测试的目的就是确认 M_I 和 M_S 的符合性,然而要确定两者的符合性,要穷举所有的输入、输出、状态转换的组合关系,这种组合数量是无限的。在实际测试过程中,尽最大努力用有限的测试序列去寻找在 M_I 错误的地方。利用 M_S 构建故障模型,依据故障模型设计出测试用例集。这套测试用例集能够识别任何包含在故障模型中的错误类型。常见的有限状态机的错误包括输出错误、迁移方向错误、迁移多余错误、迁移缺失错误、状态多余错误、状态缺失错误。

错误 1:输出错误。在 M_I 中,如果一个状态迁移产生输出和 M_S 不一致,该错误被称为输出错误。在图 5-25 中,M 为设计的状态机 M_S,而 M_1 和 M_2 分别为其不同的实现,包含不同的输出错误。在 M 中在 S_0 输入 a 时,其输出为 0,而 M_1 中在 S_0 输入 a 时,其输出为 1。在 M 中在 S_1 输入 a 时,其输出为 0,而 M_2 中在 S_1 输入 a 时,其输出为 1。

错误 2:迁移方向错误。在 M_I 中,如果一个状态迁移的源状态或者目标状态和 M_S 不一致,该错误被称为迁移错误。在图 5-26 中,M 为设计的状态机 M_S,而 M_1 和 M_2 分别为其不同的实现,包含不同的输出错误。在 M 中在 S_0 输入 a,其目标状态为 S_0,在 M 中在 S_0 输入 a,其目标状态为 S_1。在 M 中在 S_1 输入 b 时,其目标状态为 S_1,而 M_2 中在 S_1 输入 b 时,其目标状态为 S_0。

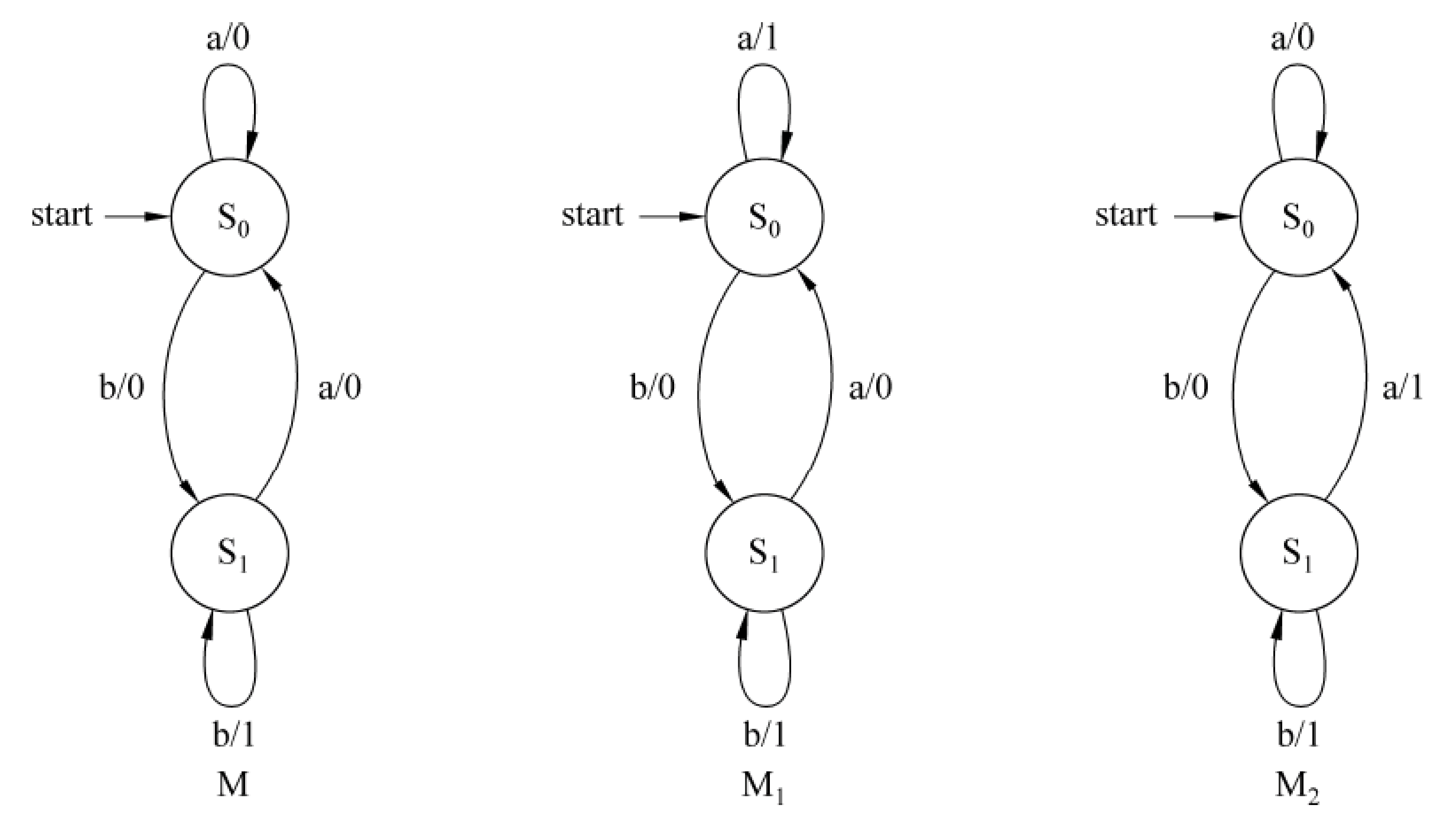


图 5-25　状态机的输出错误

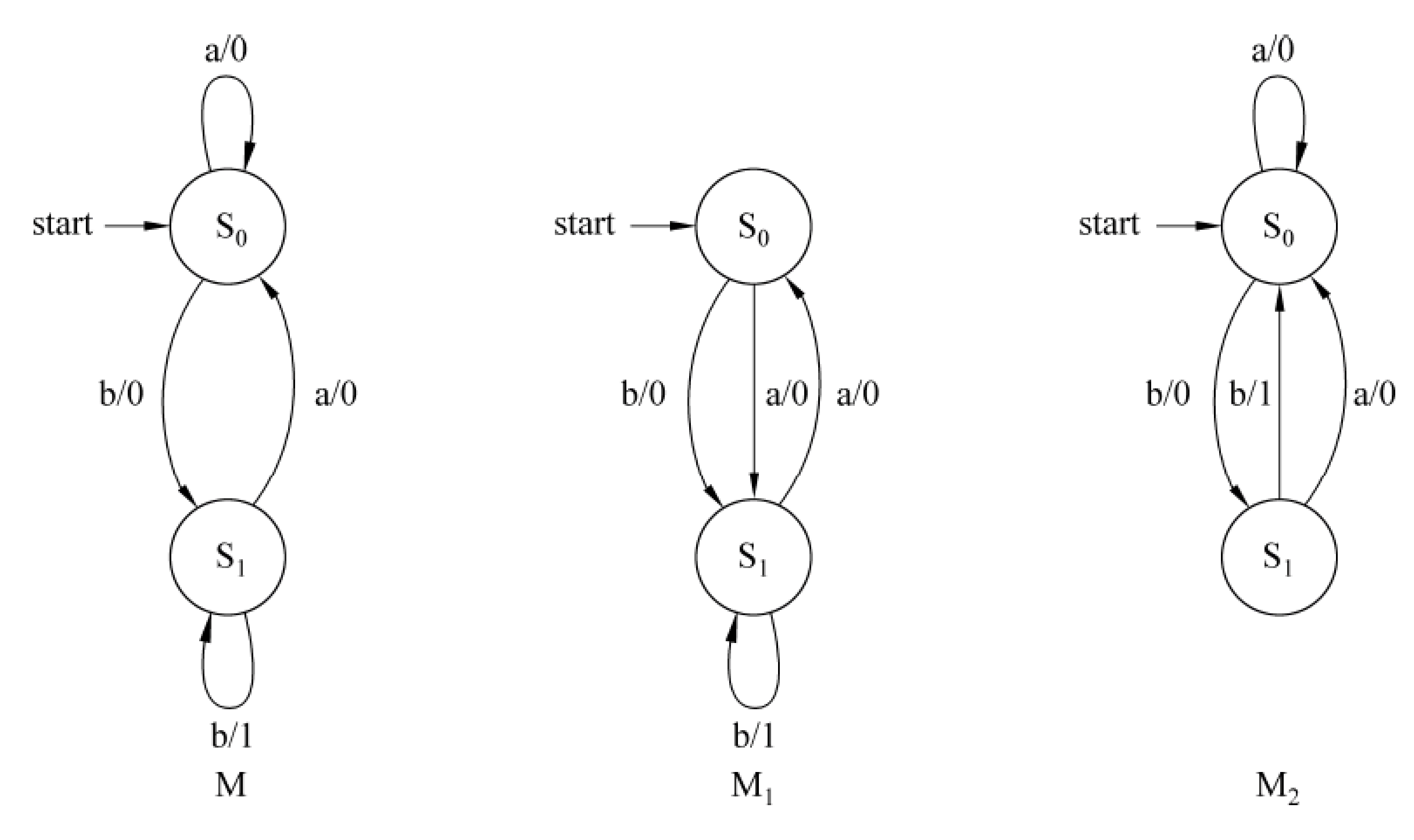


图 5-26　状态机的迁移错误

错误 3：迁移缺失错误。M_S 的不同实现 M_I，可能缺失了 M_S 定义的迁移，并导致其 M_I 和 M_S 不一致，该错误称为迁移缺失错误。

错误 4：迁移多余错误。M_S 的不同实现 M_I，可能引入了 M_S 未定义的迁移，并导致其 M_I 和 M_S 不一致，该错误称为迁移多余错误。

错误:5：状态多余错误。M_S 的不同实现 M_I，有可能引入了多余的状态，并且 M_I 和 M_S 不一致，该错误称为多余状态。在图 5-27 中，M 是设计的状态机，M 有两种状态 S_0 和 S_1。M_1 和 M_2 是 M 的两种实现，它们均有三个状态，分别是 S_0、S_1、S_2，相对于 M 而言，M_1 和 M_2 多一个状态 S_2。但是多余的状态，并不一定意味着错误。而 M_1 中，尽管相对于 M 多了一个状态 S_2，但是它是和 M 等价的。而 M_2 并不和 M 等价，而是包含不同的状态转换。

错误 6：状态缺失错误。如果 M_I 与 M_S 相比，缺失了一些状态，并且 M_I 和 M_S 不等价，该错误称为状态缺失错误。在图 5-28 中，左边为设计状态机，而右边为缺失的状态机。

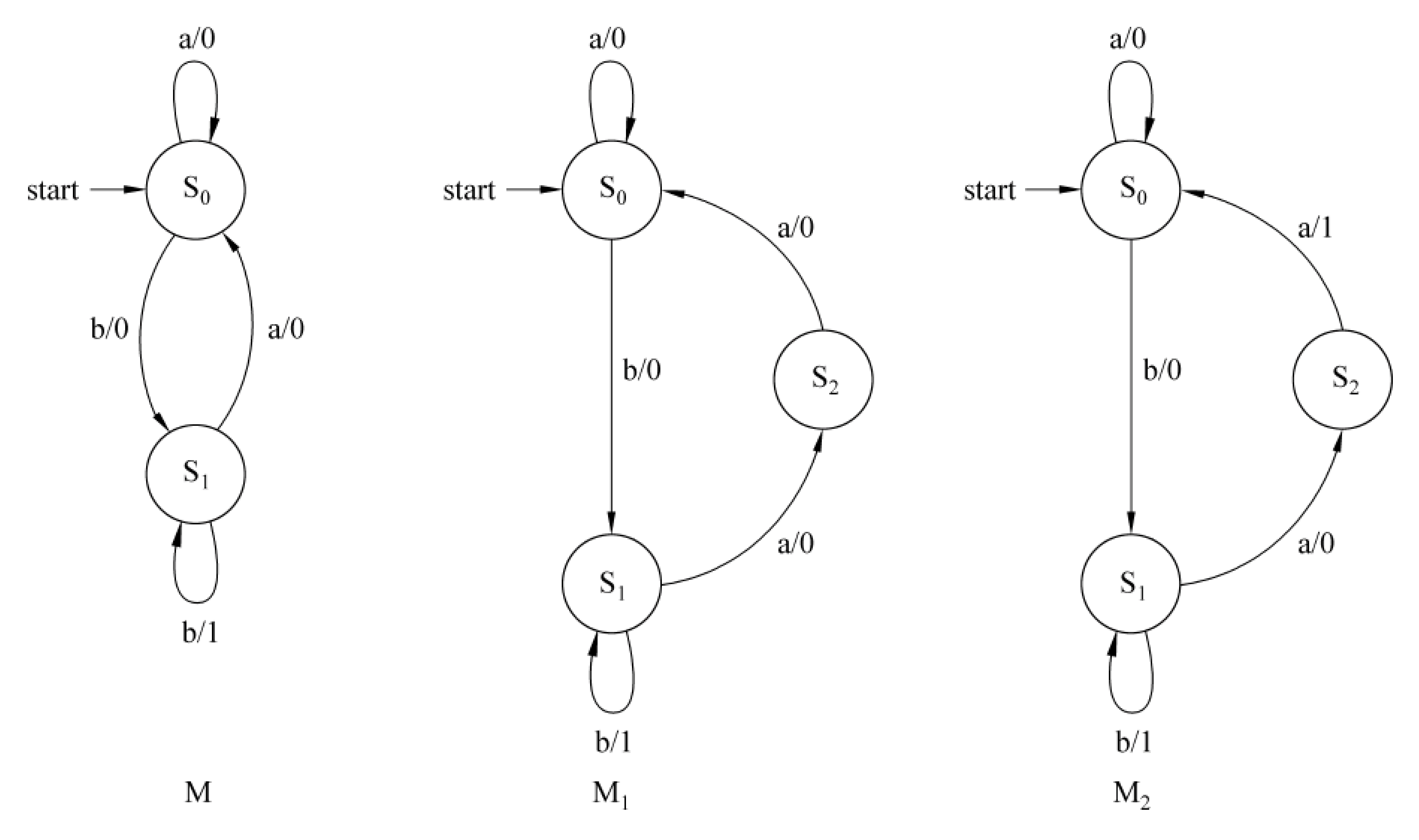


图 5-27 多余的状态

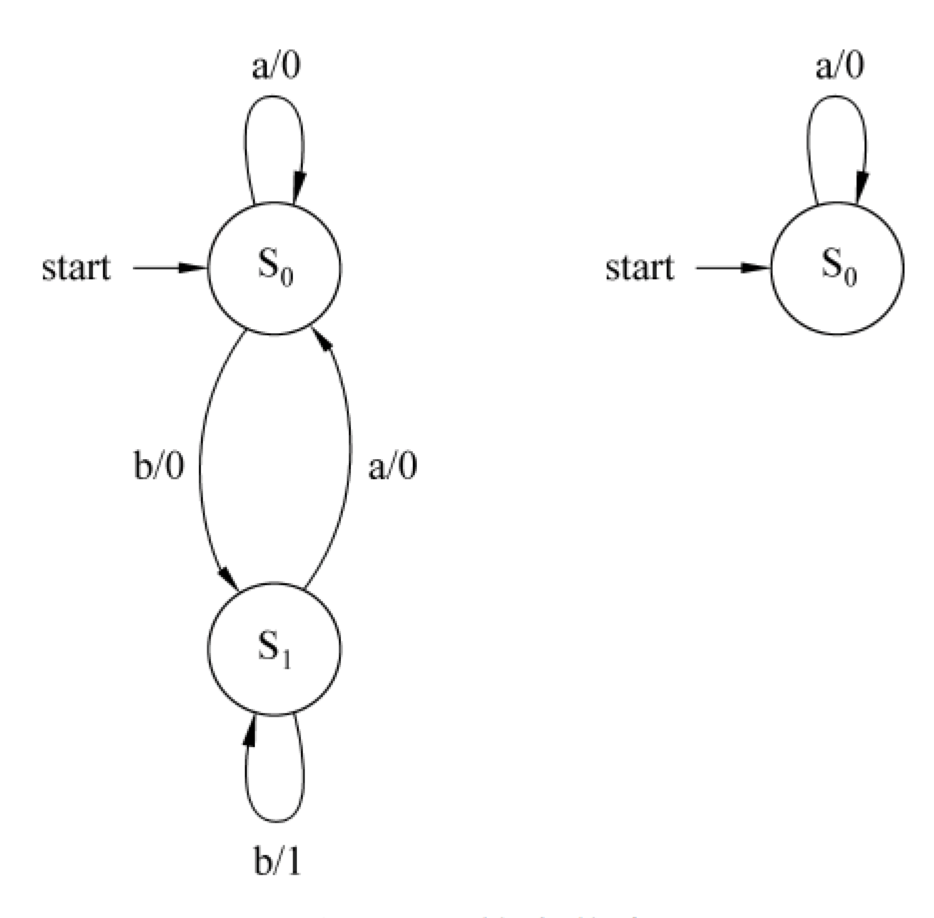


图 5-28 缺失状态

上述的错误模型可以用于评估给定有限状态机的测试用例。在实际的状态机中实现的错误情况比上述讨论的情况更加复杂。在一个状态机中，可能出现多种错误类型，也可能出现同一种类型的多个错误。

5.4 基于有限状态机的测试

5.4.1 概述

常用的基于状态机的测试包括 T 方法、D 方法、W 方法、U 方法。无论是哪种方法，本节讨论的测试方法定位于黑盒测试方法，给定一个系统的 FSM 表示 M_S，确定一个 M_I

在行为上是否和 M_S 相一致。在实际的测试过程中，可能无法获取 M_I 的实现细节，例如状态和迁移的数量。

一般的测试过程如下。

(1) 根据测试 M_S，以及相应的测试方法产生 M_S 的输入序列 S_I 和期望的输出序列 S_O，S_I 和 S_O 的成员分别来自于输入字母表 Σ 和输出字母表 Λ。

(2) 将 S_I 各个成员依次应用到 M_I。

(3) 观察 M_I 的实际输出序列 S_O'。

(4) 比较 S_O 和 S_O'，确定 M_I 和 M_S 的一致性，揭示 M_I 可能存在的一些错误，这些错误在 5.3 节中已经进行了详细的描述。

显然在实际测试过程中，根据 M_S 产生输入序列 S_I 和期望的输出序列 S_O，一般无法检测出 M_I 中存在的多余状态和迁移，因为根据 M_S 产生输入序列 S 无法假设或者推导出不存在的状态或者迁移。M_I 和 M_S 的一致性程度，根据其强度可以分为弱一致性和强一致性。若 M_I 的所有迁移和输出都和 M_S 一致，就是在 M_S 定义的范围内，两者完全一致，则称其为弱一致性。在弱一致性中，M_S 中定义的在 M_I 中均可以找到对应的内容，但不保证 M_I 中实现的自动机内容都在 M_S 中定义。强一致性，若 M_I 的所有迁移和输出都和 M_S 一致，并且两者的完备性也完全一致，则称其为强一致性。在强一致性中，在 M_I 中实现的内容，在 M_S 中能找到定义，同样地所有 M_S 中的定义均已经在 M_I 中得到了定义。

确认 M_S 中的一个迁移 $t_{ij}=(s_i, s_j, i/o)$ 是否在 M_I 中实现，通常包括以下三个步骤。

(1) 将 M_I 的状态设置成 s_i。

(2) 在 M_I 中输入字母 i，检查输出是否为 o。

(3) 检查 M_I 到达的新状态是否为 s_j。

利用步骤(2)和步骤(3)分别检查系统实现的输入错误以及迁移错误。在很多情况下，无法对系统进行精确的控制，也不是非常容易直接观察到每一个状态的输出值。也就是将状态设置成 s_i，以及确定目标状态 s_j 都有可能通过一定的输入序列才能确定。因此，在实际中确定迁移的步骤如下。

(1) 通过可选的迁移序列，将 M_I 的状态设置成 s_i。

(2) 在 M_I 中输入字母 i。

(3) 通过一个特征序列去检验 M_I 到达的新状态是否为 s_j。

5.4.2 状态覆盖测试

定义 5-1 状态覆盖集 Q

状态覆盖集 Q 是由输入序列构成，以便对于 S 中的任意状态 s_i，都存在一个输入序列 $A\in Q$，从 M_S 的初始状态 s_0 开始在输入 A 的作用下状态转移到 s_i，$\delta(s_0,A)\rightarrow s_i$。空序列 ε 属于 Q，即 $\varepsilon\in Q$，因为 $\delta(s_0,\varepsilon)=s_0$。

可以利用广度优先遍历方法构造测试树，从而得到状态覆盖集。

算法 5-1 构造状态覆盖集算法

(1) 将 s_0 作为测试树 T 的根，并将其层次标记为 1，并将 Visited[s_0]=1。

(2) 假设已经构造好了测试树 T 第 k 层($k\geqslant 1$),根据 k 层依据如下方法构造 $k+1$ 层:

① 从左到右依次选择 k 层的节点 n。

② 对于状态 n 的一个输入 a,如果 $\delta(n,a)=m$ 并且 Visited[m]=0,则将 m 作为 n 的孩子节点增加到测试树 T 中,设置 Visited[m]=1,标记从 n 到 m 的边为 a。

(3) 重复步骤(2),直到所有的状态都已经添加到测试树 T 中。

(4) 对测试树 T 中每一个节点 n,构造从根 s_0 到节点 n 所构成的边序列 A_n,将 A_n 增加到 Q 中。

例 5-7 如图 5-29 所示的状态机 M,求其状态覆盖集。

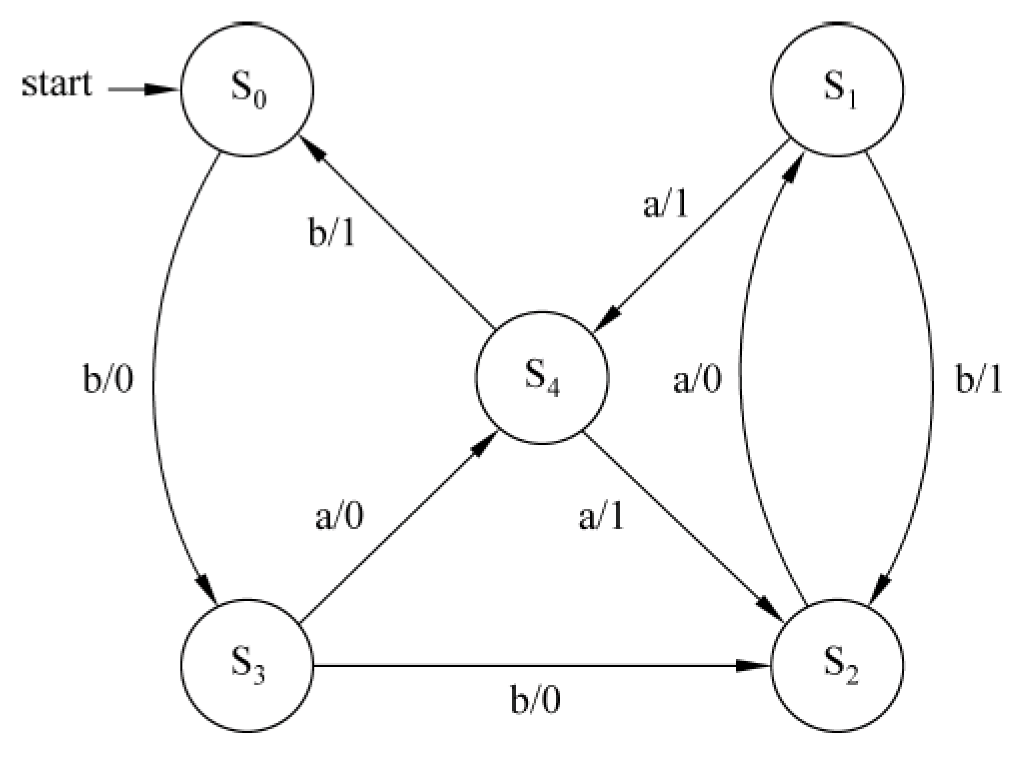


图 5-29 状态机 M

根据算法 5-1,可以构建状态覆盖集。

(1) 选择 s_0 作为测试树的根,并设置访问标志 Visited[s_0]=1。

(2) 处理第一层:s_0 只有接收一个输入 b,在 b 的作用下状态转移到 s_3,为了区分将输入 a 的作用下后继状态作为左孩子,在 b 作用下的后继状态作为右孩子,将 s_3 作为 s_0 的右孩子,同时设置 s_3 的访问标记 Visited[s_3]=1。

(3) 处理第二层:s_3 在输入 a、b 的作用下,分别转移到 s_2、s_4 两个状态。将它们添加到测试树中。

(4) 以此类推,形成最后的测试树,如图 5-30 所示。

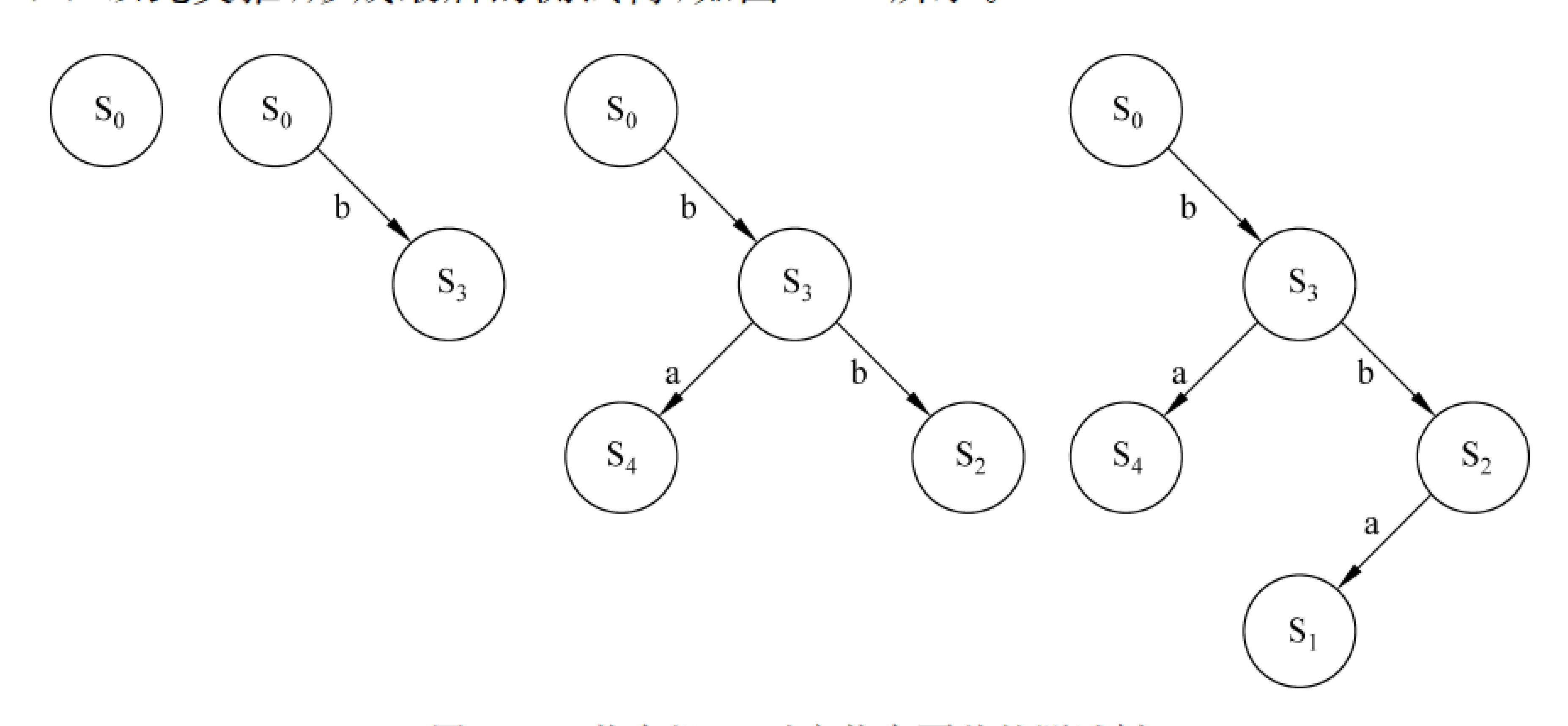


图 5-30 状态机 M 对应状态覆盖的测试树

(5) 从根节点开始,遍历每一个节点,并将形成的路径添加到 Q 中,最后形成的状态覆盖集为:Q={ε,b,ba,bb,bba}。

5.4.3 迁移覆盖测试

有限状态机 M 处于状态 s_i 时接收到输入 a∈Σ 后,产生输出 y=ω(s_i,a),并且迁移到状态 s_j=δ(s_i,a),即产生一个迁移 t,记为 t(s_i,a/y,s_j)。

在如图 5-29 所示的状态机中,包含的迁移如下:

(s_0,b/0,s_3),(s_3,b/0,s_2),(s_3,a/0,s_4),(s_4,b/1,s_0),(s_4,a/1,s_2),(s_2,a/0,s_1),(s_1,b/1,s_2),(s_1,a/1,s_4)

迁移覆盖,指有限状态机中的每一个迁移,至少被覆盖一次。在产生迁移覆盖的过程中,从初始节点开始,根据广度优先策略访问状态机的所有迁移,算法 5-2 描述了其详细过程。显然迁移覆盖必定是状态覆盖,但是由于一个状态可能有多个输入,也可能有多个输出,状态覆盖不一定能够达到变迁覆盖。

算法 5-2 构造迁移覆盖集 P 的算法

(1) 将 s_0 作为测试树 T 的根,并将其层次标记为 1。

(2) 假设已经构造好了测试树 T 第 k 层(k≥1),根据 k 层依据如下方法构造测试树的 k+1 层:

① 从左到右依次选择 k 层的节点 n;

② 如果节点 n 已经出现在 1~k-1 层的任何一层,那么节点 n 作为叶子节点,不再扩展;

③ 若节点 n 不是叶子节点,对于状态 n 的每一个输入 a,如果 δ(n,a)=m,则将 m 作为 n 的孩子节点增加到测试树 T 中,标记从 n 到 m 的边为 a。

(3) 重复步骤(2),直到所有的迁移都已经添加到测试树 T 中。

(4) 对测试树 T 中每一个节点 n,构造从根 s_0 到节点 n 所构成的边序列 A_n,将 A_n 增加到 P 中。

例 5-8 如图 5-31 所示的状态机 M,求其迁移覆盖集。

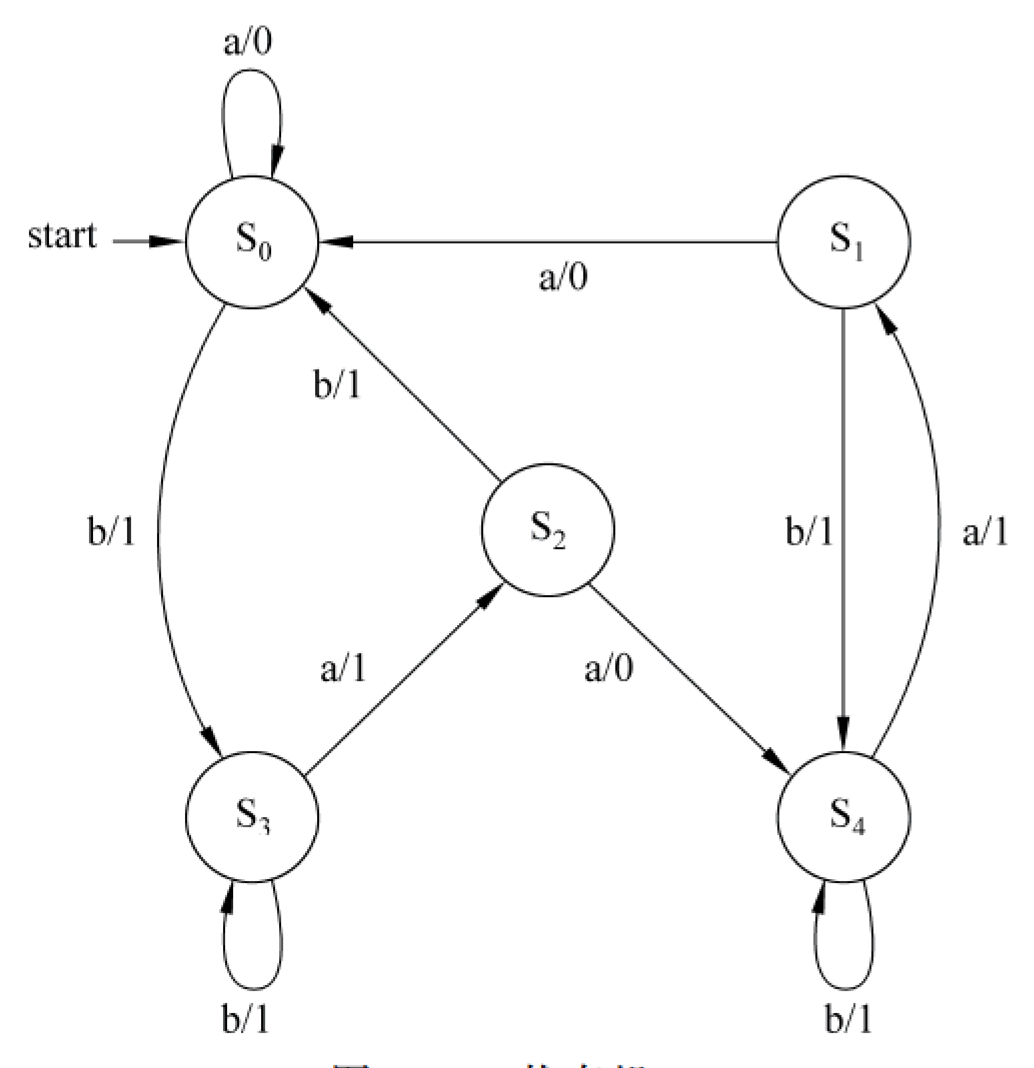


图 5-31 状态机 M

根据算法5-2描述，可以构建迁移覆盖集。

（1）选择 s_0 作为测试树的根作为第一层。

（2）处理第一层：s_0 接收输入a，由于在a的作用下，状态迁移到 s_0，s_0 作为 s_0 的左孩子。s_0 接收输入b，在b的作用下状态转移到 s_3，为了区分将输入a的作用下后继状态作为左孩子，在b作用下的后继状态作为右孩子，将 s_3 作为 s_0 的右孩子。

（3）处理第二层：s_0 由于在前面已经出现，所以 s_0 作为叶子节点不再扩展，而 s_3 扩展为 s_3，s_2 两个节点。

（4）以此类推，形成最后的测试树，如图5-32所示。

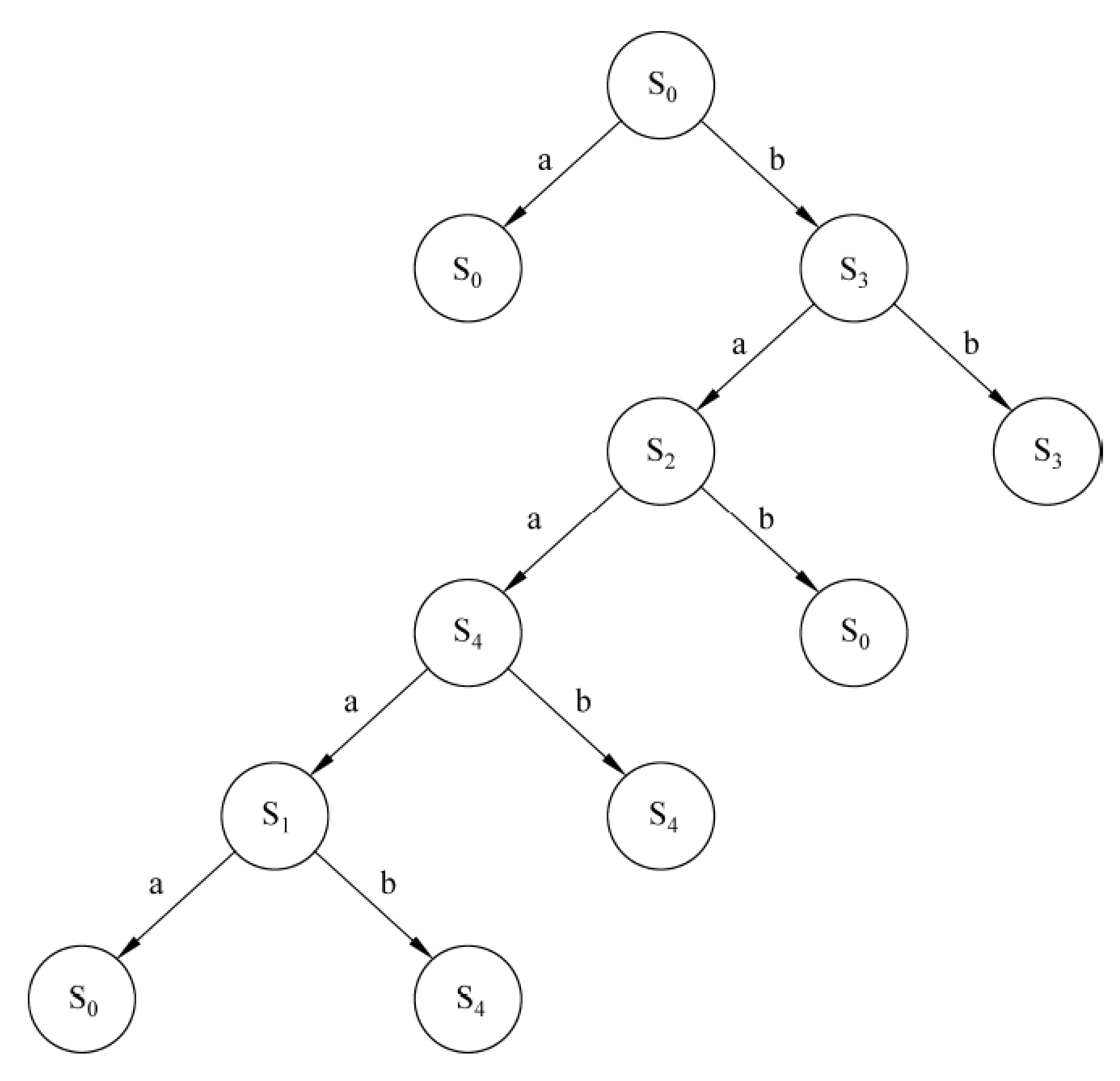


图5-32 迁移覆盖集

（5）从根节点开始，遍历每一个节点，并将形成的路径添加到P中，最后形成的迁移覆盖集为：P={ε，a，b，ba，bb，baa，bab，baaa，baab，baaaa，baaab}。

5.4.4 周游法（T方法）

周游法，又称T方法。该方法产生的输入序列，该序列有限状态机从初始状态 S_0 开始，每一个迁移至少被执行一次。T方法是最简单直接的一种测试方法，但是其并不检查 M_1 的新状态。T方法能够检查出所有的输出错误，无法保证检测出所有的迁移错误。

在T方法中，大部分采用随机产生的测试序列，直到所有的迁移均被覆盖。T方法只是和迁移覆盖基本相同。但是迁移覆盖指到达的状态是直接可以确认的。而T方法，只检查其对应输出，不对其达到的状态进行确认。

例 5-9 T 方法的测试示例。

如图 5-33(a)所示的有限状态机为和设计的有限状态机一致的实现，具有三个状态 s_0、s_1、s_2，两个输入字符 a 和 b，6 个迁移。T 方法所产生的一个测试序列为 T-sequence={a,a,a,b,b,b}，如表 5-9 所示。

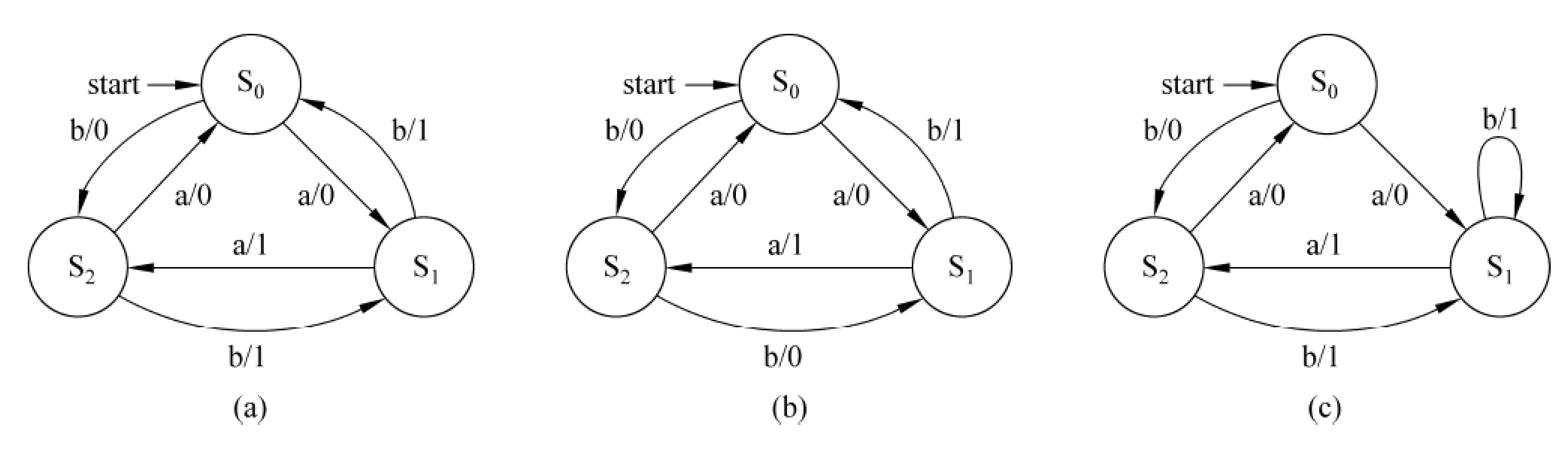


图 5-33 T 方法测试

表 5-9 T 方法产生测试序列情况

输入字母	a	a	a	b	b	b
当前状态	s_0	s_1	s_2	s_0	s_2	s_1
期望输出	0	1	0	0	1	1
实际输出	0	1	0	0	1	1

如图 5-33(b)所示的有限状态机在实现上存在一个错误，即在 s_2 到 s_1 的迁移，在输入 b 时，设计输出为 1，而实现的输出为 0。测试序列 T-sequence={a,a,a,b,b,b}，能发现该错误，如表 5-10 表中带有下划线的一列所示。

表 5-10 T 方法测试序列发现输出错误

输入字母	a	a	a	b	<u>b</u>	b
当前状态	s_0	s_1	s_2	s_0	<u>s_2</u>	s_2
期望输出	0	1	0	0	<u>1</u>	1
实际输出	0	1	0	0	<u>0</u>	1

如图 5-33(c)所示的有限状态机在实现上存在一个错误，在状态 s_1 下输入字母 b，迁移的目标状态应该为 s_0，而在该实现中，迁移的目标状态仍然为 s_1。但是。T 方法所产生的一个测试序列为 T-sequence={a,a,a,b,b,b}，由于不检查其目标状态，所以无法发现该错误。

由于 T 方法产生的测试序列是随机产生的，其产生的测试用例并不唯一。例如，以下几个测试序列均能满足要求。

T1-sequence={a,b,b,b,a,a }

T2-sequence={b,b,b,a,a,a }

T3-sequence={a,a,b,b,b,a }

同时该方法的测试输入序列中也可能存在大量的冗余。例如，以下几个测试序列均存在一定冗余，但其只能满足T方法的覆盖。

T4-sequence={a,a,a,a,a,a,b,b,b}

T5-sequence={a,a,a,a,a,a,a,a,a,b,b,b}

5.4.5 区分序列法(D方法)

首先对有限状态机构造一个区分序列(Distinguishing Sequence,DS)，然后根据该区分序列构造测试输入序列。和U方法一样，该方法只适用于存在区分序列的状态机。

区分序列：如果存在一条序列能够区分所有的状态，那么这条序列被称为区分序列。区分序列用于确认 M_I 所达到的状态是否为预期的状态。对于S中的任意两个状态，输入区分序列X，它们的输出均不相同，则称X为区分序列。也就是：

$\forall s_i, s_j \in S, s_i \neq s_j$，它们的分离序列 $X \in \Sigma^*$，使得 $\omega(s,X) \neq \omega(t,X)$。

在前面部分已经分析过：给定一个 M_S 的实现 M_I，受客观条件的限制，无法直接确定状态机所处状态。区分序列提供了一种确定 M_I 所处状态的有效途径，同时也用于黑盒方法确定 M_I 和 M_S 的一致性。

给定一个状态机 $M_S=(\Sigma, \Lambda, S, S_0, \delta, \omega)$，其中 $|S|=n$。定义状态块为S的一个多重集的组合 $\mathscr{B}$，使得 $\mathscr{B}$ 所有成员的势(或者称为基数)总和等于 n。在一个集合中，相同的元素只能出现一次，因此只能显示出有或无的属性。在多重集之中，同一个元素可以出现多次。一个元素在多重集里出现的次数称为这个元素在多重集里面的重数。对于有限多重集，势的大小可用集合的元素个数来进行度量。

例如，若一个 M_S 含有4个状态，即 $S=\{s_0,s_1,s_2,s_3\}$，表5-11给出了其部分可能的状态块，$\mathscr{B}_1$ 只包含一个多重集，其势为4。$\mathscr{B}_3$ 包含两个多重集，在第一个多重集中，包含两个不同的元素，而在第二个多重集中，包含两个相同的元素 s_0。$\mathscr{B}_6$ 中，包含四个多重集，每一个多重集仅包含一个元素，并且前两个多重集是相同的。

表5-11 状态块示例

序号	状态 $\mathscr{B}$	状态块	序号	状态 $\mathscr{B}$	状态块
1	$\mathscr{B}_1$	$\{(s_0 s_1 s_2 s_3)\}$	4	$\mathscr{B}_4$	$\{(s_0),(s_1),(s_1),(s_1)\}$
2	$\mathscr{B}_2$	$\{(s_0 s_1),(s_2 s_3)\}$	5	$\mathscr{B}_5$	$\{(s_0),(s_1),(s_2),(s_3)\}$
3	$\mathscr{B}_3$	$\{(s_0 s_1),(s_2 s_2)\}$	6	$\mathscr{B}_6$	$\{(s_0),(s_0),(s_2),(s_3)\}$

给定状态机的一个状态块 $\mathscr{B}=\{W_1,W_2,\cdots,W_i,\cdots,W_m\}$，以及一个输入a，其中 $a\in\Sigma$，定义转换函数 T_r：$\mathscr{B}'=T_r(\mathscr{B},a)$。对于 $\mathscr{B}$ 的一个成员 $W_i=(W_{i1},W_{i2},\cdots,W_{ik})$，根据以下规则获得一个或者多个 $\mathscr{B}'$ 成员。

(1) 若 $\omega(W_{ij},a)=\omega(W_{ik},a)$，且 $\delta(W_{ij},a)=W'_{ij}$，$\delta(W_{ik},a)=W'_{ik}$，那么 W'_{ij} 和 W'_{ik} 在同一个集合中。

(2) 若 $\omega(W_{ij},a)\neq\omega(W_{ik},a)$，且 $\delta(W_{ij},a)=W'_{ij}$，$\delta(W_{ik},a)=W'_{ik}$，那么 W'_{ij} 和 W'_{ik} 不

在同一个集合中。

现在，可以根据转换函数 T_r 构造一个区分序列 DS 树。首先，构造一个状态块作为 DS 树的树根，该状态块仅包含一个多重集，且该多重集包含所有的 M_S 状态，将 DS 树根记为第 1 层。接着，对于 DS 树第 i 层($i \geqslant 1$)的所有状态块应用转换函数 T_r 来获得第 $i+1$ 层的状态块。DS 树的每一节点最多具有$|\Lambda|$个孩子，Λ 为 M_S 的输出字母的集合。理论上，由于 M_S 可能存在的回路性质，DS 树的层数可能是无限的。无限层数的 DS 树对于测试执行是不现实的，所以对于 DS 树的层数一般是有限制的。满足下面三个条件之一的节点不再进行扩展。

(1) 存在同质状态块(记为 D_1)。若一个状态块 $\mathscr{B}$ 的某一个成员 W_i 具有相同的元素，则称状态块 $\mathscr{B}$ 为同质的状态块。在表 5-11 中的 $\mathscr{B}_3$ 的第二个成员($s_2 s_2$)具有相同的元素 s_2，所以 $\mathscr{B}_3$ 是同质的状态块。

(2) 存在单例状态块(记为 D_2)。若一个状态块 $\mathscr{B}$ 的所有成员 W_i 均只有一个元素，则称状态块 $\mathscr{B}$ 为单例状态块。在表 5-11 中的 $\mathscr{B}_4$，$\mathscr{B}_5$，$\mathscr{B}_6$ 均为单例状态块，它们所有的成员均只有一个元素。

(3) 存在重复状态块(记为 D_3)。若从 DS 根(初始状态块)到当前的状态块的路径上，已经存在和当前状态块相同的状态块，则称为重复状态块。

在扩展过程中，如果出现了单例状态块，意味着从根节点(初始状态块)开始，到单例状态块所经过的路径所有的输入字母构成了区分序列 DS。如果无法形成单例状态块，也就意味着无法成功构造区分序列 DS。对于同质状态块，意味着无法通过扩展到达单例状态块。所以 M_S 存在区分序列 DS，当前仅当在 DS 树上存在单例状态块。

算法 5-3　构造区分树 DS

(1) 构造初始状态块 $\mathscr{B}$，并将其作为 DS 的根。初始 $\mathscr{B}$ 仅包含一个成员，该成员的元素由 M_S 的所有状态所构成。将初始状态块 $\mathscr{B}$ 添加到状态队列中。

(2) 当状态块队列非空，循环执行：

① 对于 DS 的 $i(i \geqslant 1)$层，查找非终止状态块 $\mathscr{B}$：

若找到非终止状态块 $\mathscr{B}$，将其移出队列，并应用转换函数 $T_r(\mathscr{B}, a)$计算新一层的状态块 $\mathscr{B}'$，其中 $a \in \Sigma$。否则，算法终止。

② 将步骤①中产生的状态块 $\mathscr{B}'$添加到 DS 树中，在步骤①中产生的状态块 $\mathscr{B}$ 和本状态块 $\mathscr{B}'$的分支上标记输入字母 a，判断其是否满足 D_1，D_2，D_3。若满足三个条件中的任意一个，则将其标记为终止状态块。否则将状态块 $\mathscr{B}'$添加到状态块队列中。

(3) 若存在单例状态块，则其从根节点到单例状态块路径上的所有输入构成了区分序列。否则表明区分序列不存在。

并非所有的 M_S 都存在 DS。在形成了 DS 树以后，如果所有的叶子节点(终止)都不是单例状态块，那么该 M_S 不再存在 DS。

例 5-10　求图 5-34 中状态机的 DS。

在图 5-34 中，共有 4 个状态，每个状态接收两个输入，字母 a 和 b，输出集合为数字 0 和 1。首先构造初始的状态块 $\mathscr{B}$，其包含一个元素(s_0，s_1，s_2，s_3)。将 $\mathscr{B}$ 添加到队列中。

从队列中弹出状态块 $\mathscr{B}$，对于输入字母 a，产生状态块 $\mathscr{B}_1=\{(s_0, s_1), (s_3, s_3)\}$，因为

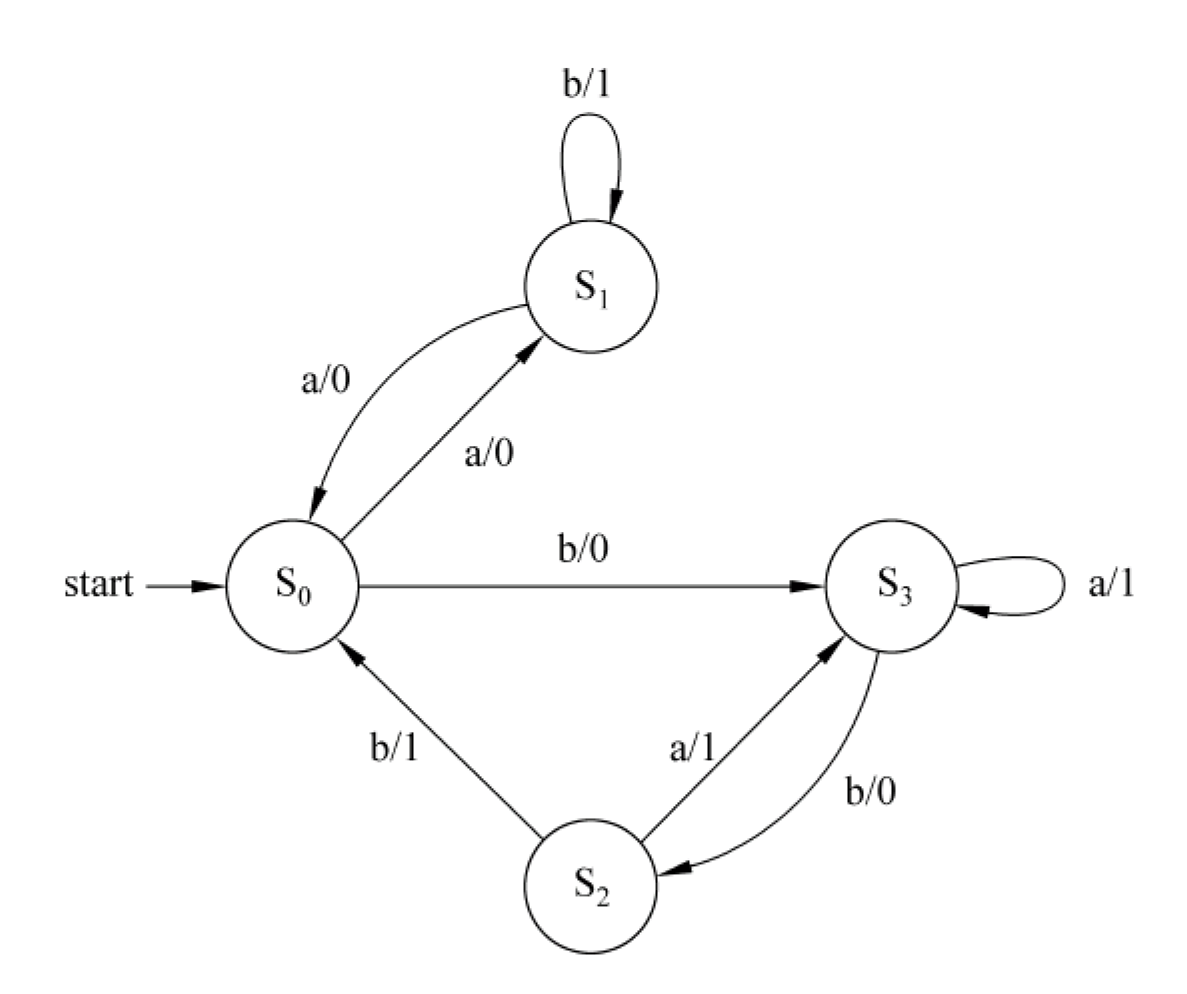


图 5-34　有限状态机

状态 s_0 和 s_1 产生同样的输出 0，并且状态机转换到状态 s_1 和 s_0，状态 s_2 和 s_3 产生同样的输出 1，并且状态机转换到状态 s_3。由于 $\{(s_0,s_1),(s_3,s_3)\}$ 中的第二个元素存在两个 s_3，所以 $\mathscr{B}_1$ 为同质状态块。

状态块 $\mathscr{B}$，对于输入字母 b，产生状态块 $\mathscr{B}_2=\{(s_3,s_2),(s_1,s_0)\}$。这是因为对于状态块 $\mathscr{B}$，在输入字母 b 的作用下，状态 s_0 和 s_3 产生同样的输出 0，同时状态机分别转换到 s_3 和 s_2，状态 s_1 和 s_2 产生同样的输出 1，同时状态机分别转换到 s_1 和 s_0。将状态块 $\mathscr{B}_2$ 添加到队列中。

从队列中弹出状态块，也就是状态块 $\mathscr{B}_2$。利用同样的方法，状态块 $\mathscr{B}_2$ 在输入字母 a 的作用下，获得新状态块 $\mathscr{B}_3=\{(s_3,s_3),(s_1,s_0)\}$。同样，$\mathscr{B}_3=\{(s_3,s_3),(s_1,s_0)\}$ 也是同质状态块，并不添加到队列中。状态块 $\mathscr{B}_2$ 在输入字母 b 的作用下，获得新状态块 $\mathscr{B}_4=\{(s_0),(s_1),(s_2),(s_3)\}$，状态块 $\mathscr{B}_4$ 为单例状态块。

从初始状态块 $\mathscr{B}$ 到状态块 $\mathscr{B}_4$ 路径上所有输入序列构成了 DS=bb，如图 5-35 所示。

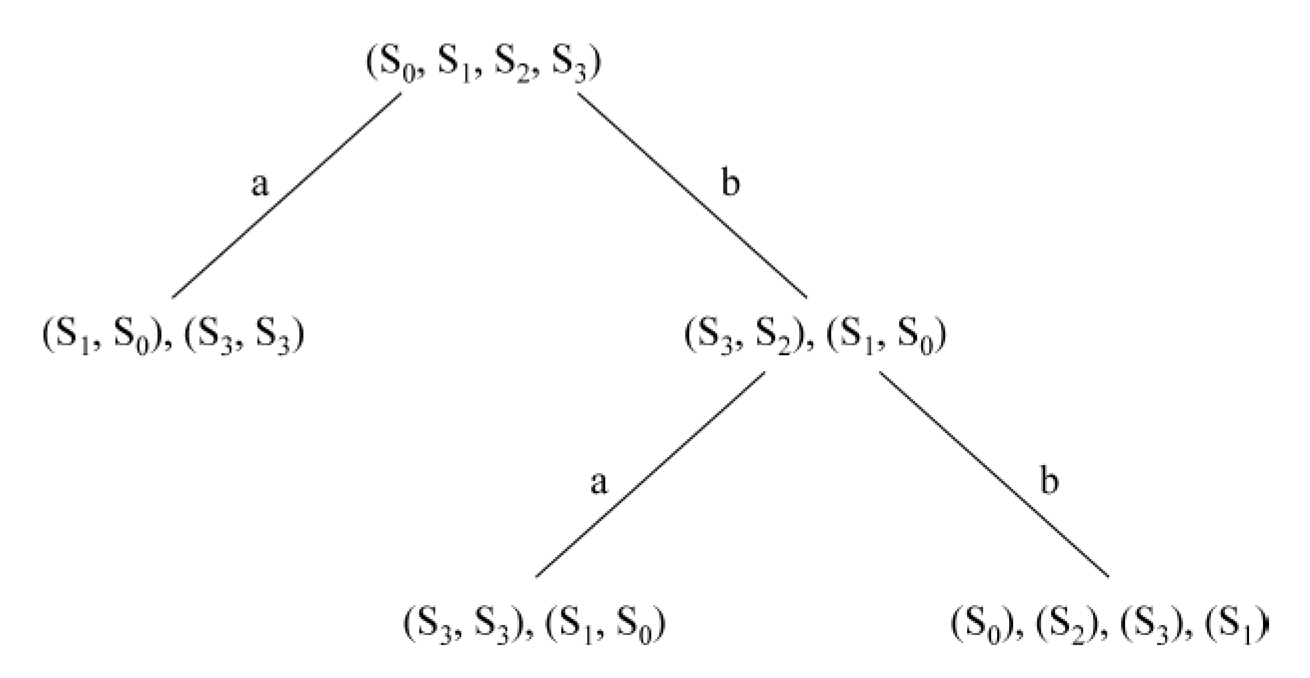


图 5-35　有限状态机对应 DS 树

在不同的状态下，接收区分序列输入所产生的输出如表 5-12 所示，s_0 在 DS 的作用下产生了输出序列 00，s_1 在 DS 的作用下产生了输出序列 11，以此类推，可以根据 DS 作用的不同输出从而区分不同的状态。

表 5-12　不同状态在 DS=ab 作用下的输出

当前状态	输出序列	当前状态	输出序列
s_0	00	s_2	10
s_1	11	s_3	01

例 5-11　不存在 DS 的有限状态机，如图 5-36 所示。

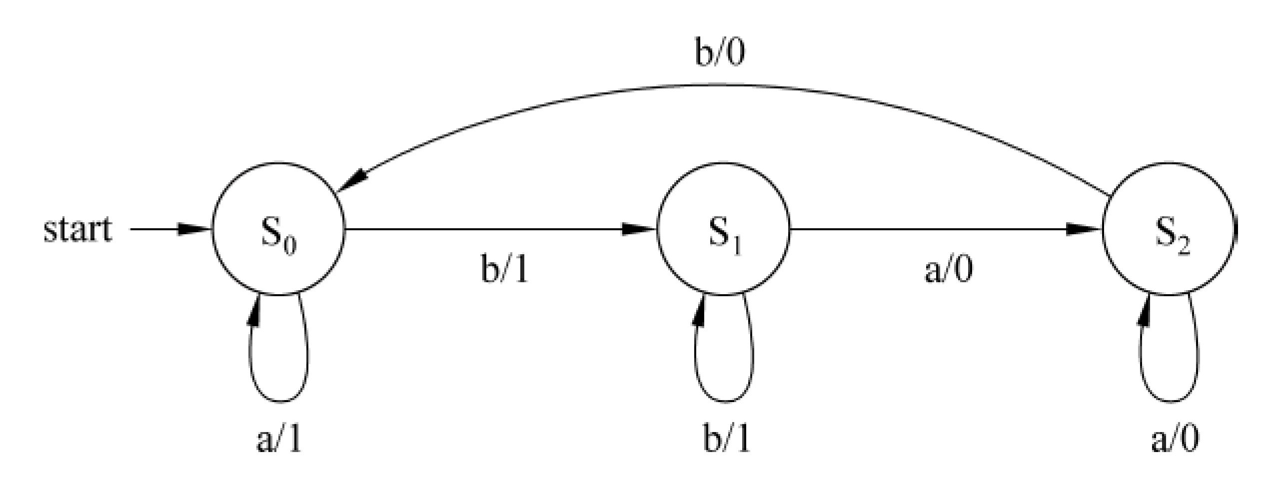
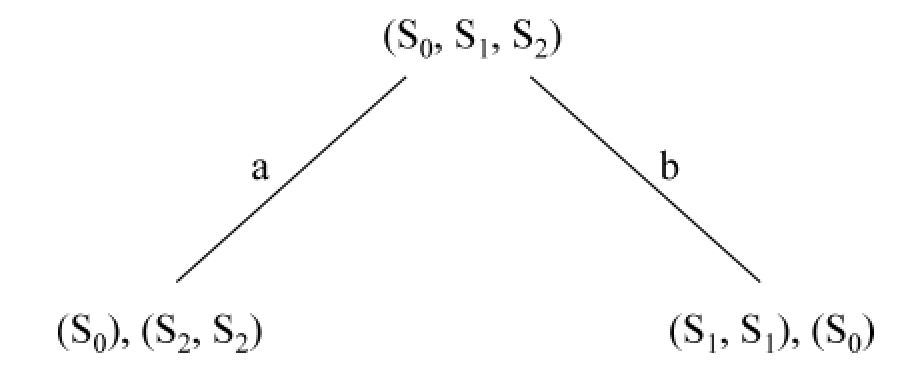


图 5-36　不存在 DS 的有限状态机

构建初始状态块 $\mathscr{B}=\{(s_0,s_1,s_2)\}$，$\mathscr{B}$ 仅包含一个成员，该成员包含 M_S 的所有状态，并将其添加到队列中。

从队列中弹出状态块 $\mathscr{B}=\{(s_0,s_1,s_2)\}$，对状态块 $\mathscr{B}$ 应用转换函数 T_r。M_S 的输入包括字母 a 和 b。由 $T_r(\{(s_0,s_1,s_2)\},a)$ 可以得到状态块 $\mathscr{B}_1=\{(s_0),(s_2,s_2)\}$，由 $T_r(\{(s_0,s_1,s_2)\},b)$ 可以得到状态块 $\mathscr{B}_2=\{(s_1,s_1),(s_0)\}$。由于新产生的 $\mathscr{B}_1$ 和 $\mathscr{B}_2$ 都是同质状态块，无法进行扩展，故该有限状态机不存在 DS。



例 5-12　存在多个 DS 的状态机，如图 5-37 所示。

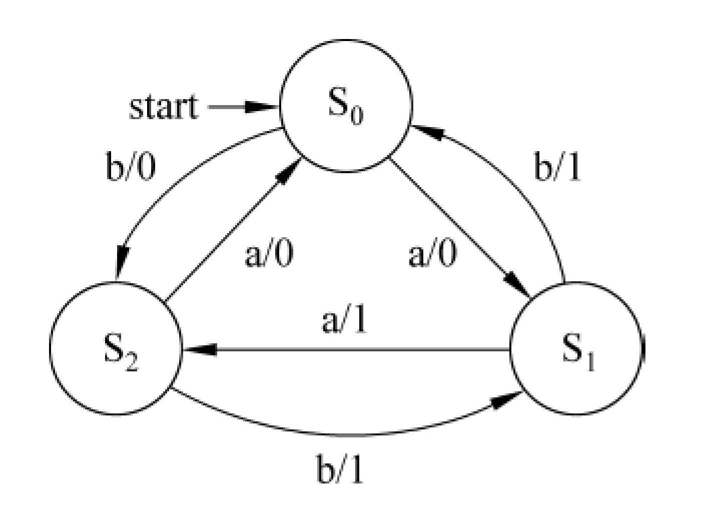


图 5-37　存在多个 DS 的有限状态机

图 5-37 中的 M_S，具体可以通过如下步骤构建 M_S 的 DS 树。

构建初始状态块 $\mathscr{B}=\{(s_0,s_1,s_2)\}$，$\mathscr{B}$ 仅包含一个成员，该成员包含 M_S 的所有状态。并将其添加到队列中。

从队列中弹出状态块 $\mathscr{B}=\{(s_0,s_1,s_2)\}$，对状态块 $\mathscr{B}$ 应用转换函数 T_r。这时候的输入包括字母 a 和 b。由 $T_r(\{(s_0,s_1,s_2)\},a)$ 可以得到状态块 $\mathscr{B}_1=\{(s_1,s_0),(s_2)\}$，由 $T_r(\{(s_0,s_1,s_2)\},b)$ 可以得到状态块 $\mathscr{B}_2=\{(s_2),(s_0,s_1)\}$。由于这两个状态 $\mathscr{B}_1$ 和 $\mathscr{B}_2$ 均加入到队列中，在 $\mathscr{B}_1$ 和 $\mathscr{B}$ 之间分支上添加输入字母 a，在 $\mathscr{B}_2$ 和 $\mathscr{B}$ 之间分支上添加输入字母 b。根节点处理完毕。

接着，处理 DS 第一层。从队列中弹出 $\mathscr{B}_1$，并应用转换函数 $T_r\{\mathscr{B}_1,a\}$ 得到 $\mathscr{B}_3=\{(s_0),(s_2),(s_1)\}$。由于 $\mathscr{B}_3$ 为单例状态块，所以将其标记为终止。同理，由 $T_r\{\mathscr{B}_1,b\}$、

$T_r\{\mathscr{B}_2, a\}$、$T_r\{\mathscr{B}_2, b\}$得到了类似的单例状态块,并将其标记为终止节点。也就是 DS 的叶子节点,如图 5-38 所示。

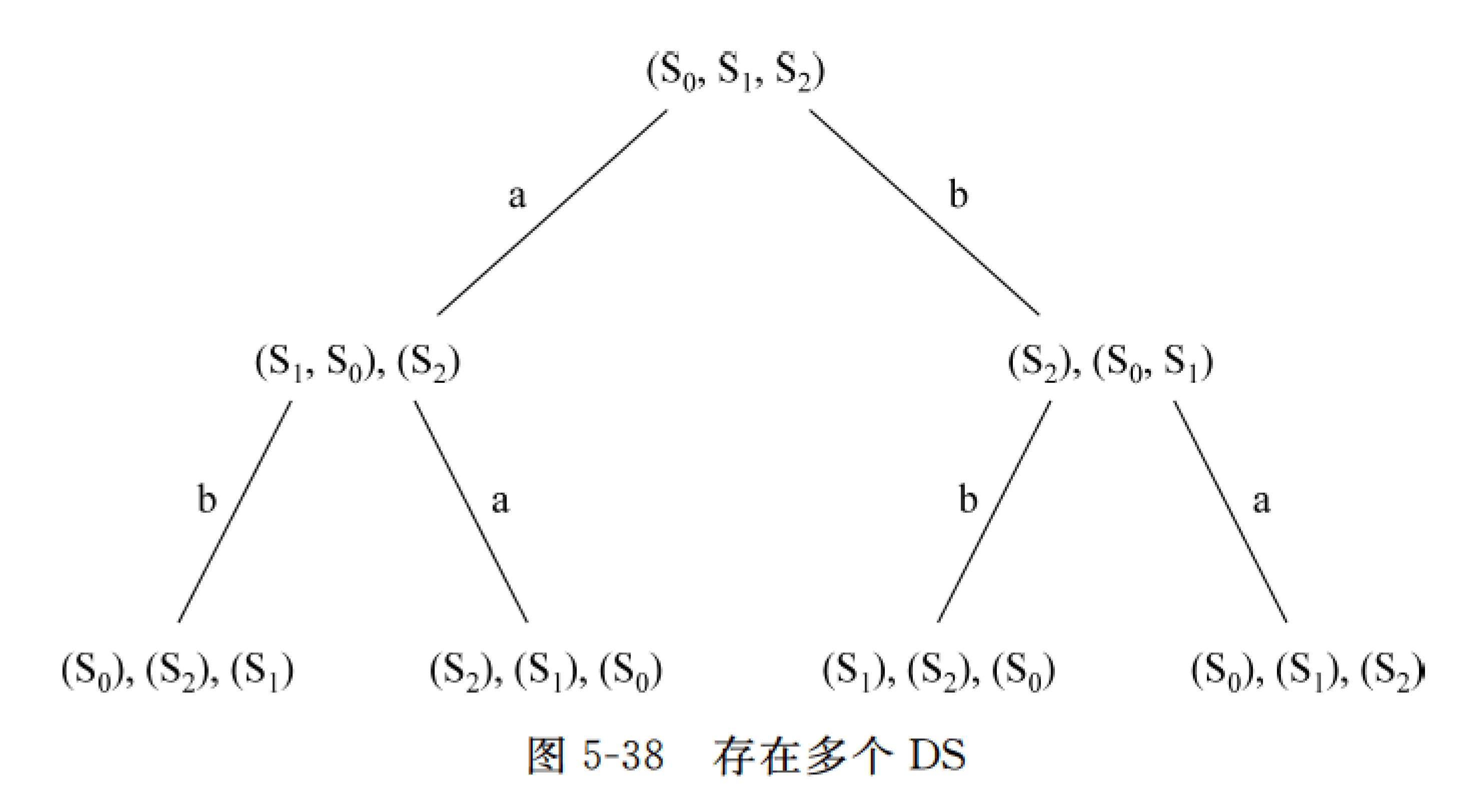


图 5-38 存在多个 DS

在形成 DS 树以后,从根节点开始经过若干中间节点到达单例状态块叶子节点的路径上所有的输入所构成的序列构成了 DS。在本例中,存在 aa,bb,ba,ab 4 个 DS。

初始状态的输入序列作用下产生的不同输出见表 5-13。

表 5-13 初始状态的输入序列作用下产生的不同输出

输入序列	初始状态	输出
aa	s_0	01
	s_1	10
	s_2	00
bb	s_0	01
	s_1	10
	s_2	11
ba	s_0	00
	s_1	10
	s_2	11
ab	s_0	01
	s_1	11
	s_2	00

5.4.6 特征序列法(W 方法)

D 方法采用区分序列 DS 一次性区分所有状态,但是由于其约束太强,很多 M_S 并不存在 DS 序列。例 5-12 中 M_S 并不存在区分序列 DS。W 方法定了特征序列(Characterizing Sequences,CS),一个 CS 也就是一个部分 DS(partial DS)。CS 可以从一

个 S 的子集中区分出一个状态 s_i，而不是区分所有状态。其基本思路是通过迭代的方法寻找 CS 的集合，以区分不同的状态。

（1）通过一个输入序列，将 M_S 的状态集划分成几个块。

（2）接着构造新的输入序列，将每一个块进一步划分成几个可以区分的子块，递归执行，直到每一个子块仅包含一个状态。

对于一个 M_S 而言，所有 CS 构成的集合，称为 M_S 的特征集 W。M_S 的特征集 W 是由输入序列构成的有限集合，对 S 中的任意状态 s_i 和 s_j，W 中都存在一个输入序列 W_k 使得 $\omega(s_i, W_k) \neq \omega(s_j, W_k)$。形式描述为 $\forall s_i, s_j \in S, \exists W_k \in W$，满足 $\omega(s_i, W_k) \neq \omega(s_j, W_k)$。对于 W 中的每一个 W_k，其长度均应该是有限的。

在如图 5-39 所示的有限状态机中，根据 5.4.5 节所讨论的方法，可以知道其区分序列 DS 并不存在。以输入字母 a 开始，s_2 和 s_3 的输出均为 1，并且转换到状态 s_3，也意味着其产生同质状态块；若以输入字母 b 开始，所有的输出均为 0，并且转换的新状态块和初始状态块相同，也就是出现重复状态块。

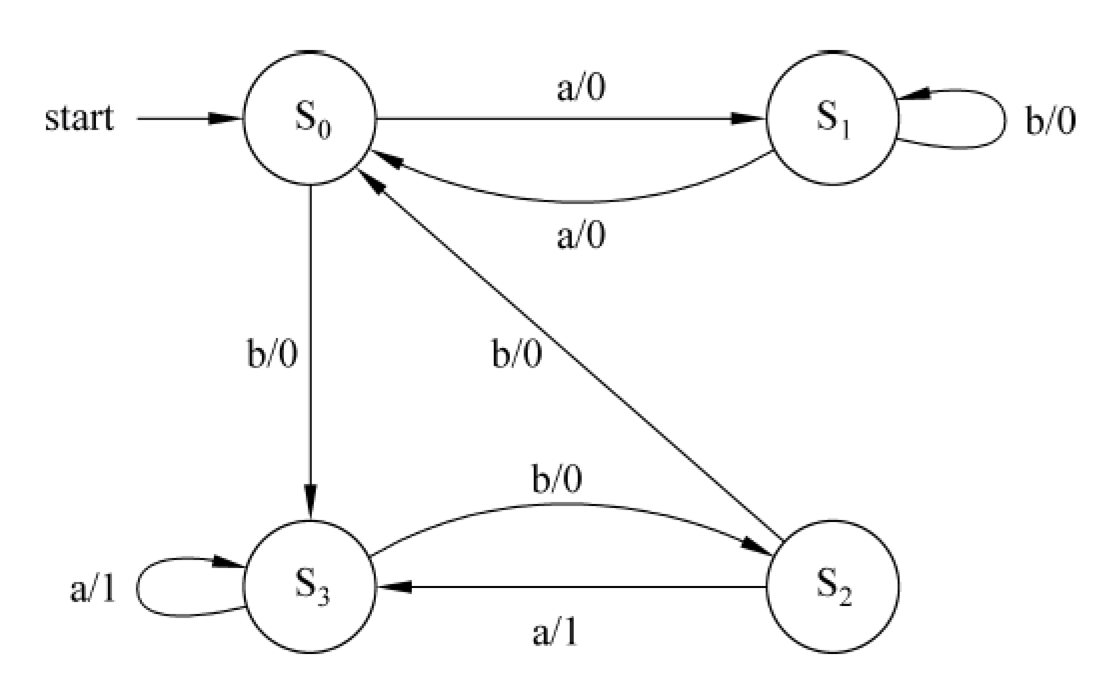


图 5-39　不存在 DS 但是存在 CS 的有限状态机

但是其存在特征集 W＝{ba，a}，如表 5-14 所示。因为 $\omega(s_1, ba) = \omega(s_2, ba) = 00$，$\omega(s_0, ba) = \omega(s_3, ba) = 01$，所以 ba 能够正确区分 (s_0, s_1)、(s_0, s_3)、(s_2, s_1)、(s_2, s_3)。在此基础上，$\omega(s_0, a) = \omega(s_1, a) = 00$，$\omega(s_2, a) = \omega(s_3, a) = 1$。

表 5-14　有限自动机的特征集

状态	特征序列 CS_1	输出	特征序列 CS_2	输出
s_0	ba	01	a	0
s_1	ba	00	a	0
s_2	ba	00	a	1
s_3	ba	01	a	1

依据上述分析，可以得到不同状态之间的区分，如表 5-15 所示。

表 5-15 状态对和特征序列

序号	状态对	特征序列	序号	状态对	特征序列
1	s_0, s_1	ba	4	s_1, s_2	a
2	s_0, s_2	ba	5	s_1, s_3	ba
3	s_0, s_3	a	6	s_2, s_3	ba

为了构造特征集，需要采用状态的 K 阶等价划分。若两个状态 $s_i, s_j \in S$，任意给定长度为 K 的输入，均产生同样的输出，则称 s_i, s_j 是 K 阶等价（K-equivalence）。有限状态机等价，对于给定的有限状态机 M_1 和 M_2，若满足以下两个条件，则称它们是等价的有限状态机。

（1）对于 M_1 任意一个状态 s，在 M_2 中存在一个对应的状态 s′，s 和 s′等价。

（2）对于 M_2 任意一个状态 s，在 M_1 中存在一个对应的状态 s′，s 和 s′等价。

M_S 状态 S 的 K 等价划分 P_k，是 n 个有限集合 $\Pi_{k1}, \Pi_{k_2}, \Pi_{k3}, \cdots, \Pi_{kn}$，并满足如下条件：

（1）$\bigcup_{i=1}^{n} \Pi_{ki} = S$。

（2）$\forall s_i, s_j \in \Pi_{km}$，$s_i$ 和 s_j 满足 K 等价。

（3）$\forall s_i \in \Pi_{km}, \forall s_j \in \Pi_{kl}, m \neq l$，那么 s_i 和 s_j 是 K 可区分的。

以如图 5-31 所示的状态机为例，介绍 K 阶等价划分的方法。该状态机所对应的状态转换表如表 5-16 所示。

表 5-16 状态转换表

当前状态	输出		后继状态	
	a	b	a	b
s_0	0	1	s_0	s_3
s_1	0	1	s_0	s_4
s_2	0	1	s_4	s_0
s_3	1	1	s_2	s_3
s_4	1	1	s_1	s_4

根据状态转换表，前三个状态$\{s_0, s_1, s_2\}$在输入 a 时，其输出均为 0，而输入 b 时，其输出均为 1。同理，后两个状态$\{s_3, s_4\}$，在输入 a 时，其输出均为 1，而输入 b 时，其输出均为 1。前三个状态$\{s_0, s_1, s_2\}$，后两个状态$\{s_3, s_4\}$分别是 1-等价的。$\Pi_{11} = \{s_0, s_1, s_2\}$，$\Pi_{12} = \{s_3, s_4\}$。

为了后继讨论的方便，重新整理状态转换表，添加等价集合编号，在后继状态添加一个新的下标表示其所在集合。例如，s_0 的两个后继分别为 s_0 和 s_3，由于 s_0 处于编号为 1 的 1-等价集合中，所以添加第二个下标 1。s_3 处于编号为 2 的 1-等价集合中，所以添加第二个下标 2，所以它们的新编号分别为 s_{01} 和 s_{32}。对所有的状态，做同样的处理，行成了 1 阶等价表，标记为 P_1 表，如表 5-17 所示。

表 5-17　状态机的 1 阶等价表 P_1

1 阶等价集合	当前状态	后继状态	
		a	b
1	s_0	s_{01}	s_{32}
	s_1	s_{01}	s_{42}
	s_2	s_{42}	s_{01}
2	s_3	s_{21}	s_{32}
	s_4	s_{11}	s_{42}

接下来以 1 阶等价表 P_1 为基础构造 2 阶等价表 P_2。在等价表 P_1 中，在当前同一组内，意味着在当前长度 1 的输入字母中，其输出都是相同的，而第二个下标相同，意味在接收当前输入以后的后继状态也将在同一组内，也就意味着输入长度为 2 的序列也将产生相同的输出。因此，将所有第二个下标相同的组成一组，进一步划分形成 2 阶等价表。在 P_2 表中，s_0 和 s_1 在长度为 2 的小组输出均相同，例如，s_0 和 s_1、s_3 和 s_4 在输入长度为 2 的情况如表 5-18 所示。

表 5-18　P2 表中长度为 2 的情况

输入	aa	ab	ba	bb
s_0	00	01	11	11
s_1	00	01	11	11
s_3	10	11	11	11
s_4	10	11	11	11
s_2	10	11	10	11

在 P_1 等价表中，s_0 和 s_1 两个状态在输入 a 的作用下，其后继状态的第二个下标都是 1，而在输入 b 的作用下，其后继状态的第二个下标都是 2。s_2 状态在输入 a 的作用下，其后继状态的第二个下标是 2，而在输入 b 的作用下，其后继状态的第二个下标是 1，和 s_0 和 s_1 两个状态并相同。将 s_0 和 s_1 两个状态划分成新的一组，这样将输入分成了三个 2 阶等价集合：$\Pi_{21}=\{s_0,s_1\}$，$\Pi_{22}=\{s_2\}$，$\Pi_{23}=\{s_3,s_4\}$，如表 5-19 所示。

表 5-19　状态机的 2 阶等价表 P_2

2 阶等价集合	当前状态	后继状态	
		a	b
1	s_0	s_{01}	s_{33}
	s_1	s_{01}	s_{43}
2	s_2	s_{43}	s_{01}
3	s_3	s_{22}	s_{33}
	s_4	s_{11}	s_{43}

在等价表 P_2 中，在当前同一组内，意味着在当前长度 2 的输入字母中，其输出都是相同的，而第二个下标相同，意味在接收当前输入以后的后继状态也将在同一组内，也就意味着输入长度为 3 的序列也将产生相同的输出。第一组的两个下标完全相同，在长度为 3 的输入下的输出完全相同，如表 5-20 所示。

表 5-20 在第一组下标相同时，长度为 3 的输出相同

输入	aaa	aab	aba	abb	bab	bbb
s_0	000	001	011	011	111	111
s_1	000	001	011	011	111	111

因此，将 P_2 表中第二个下标相同的划分为一组，构造 3 阶等价表 P_3。在例子中，第一组第二个下标完全相同，而第三组第二个下标出现不同，所以将第二组划分为两个小组，构成了新的小组，如表 5-21 所示。

表 5-21 状态机的 3 阶等价表 P_3

3 阶等价集合	当前状态	后继状态	
		a	b
1	s_0	s_{01}	s_{33}
	s_1	s_{01}	s_{44}
2	s_2	s_{44}	s_{01}
3	s_3	s_{22}	s_{33}
4	s_4	s_{11}	s_{44}

以此类推，将第一组继续划分新的小组，最后形成 P_4 表，如表 5-22 所示。

表 5-22 状态机的 4 阶等价表 P_4

3 阶等价集合	当前状态	后继状态	
		a	b
1	s_0	s_{01}	s_{34}
2	s_1	s_{01}	s_{45}
3	s_2	s_{45}	s_{01}
4	s_3	s_{23}	s_{34}
5	s_4	s_{12}	s_{45}

在该表中，每一个状态都处在不同的集合中，或者说每一个集合仅有一个状态，不存在两个无法区分的状态。在 W 方法中，必须要求状态机是最简的，其目的就是为了避免出现不可区分状态，否则这个算法是不收敛的，无法找到一个有限长输入序列区分两个不同的状态。

由于阶数越高，表示要区分在同一组的状态的输入序列越长，所以根据以上构建等价表划分的逆过程，可以构造出特征序列。构造 s_i 和 s_j 的特征序列 CS 的算法如下。

算法 5-4　构造特征序列

(1) 查找等价表 P_k 和 P_{k+1}（$1 \leqslant k \leqslant |S|$），使得（$s_i$，$s_j$）在 P_k 中的同一个组中，而在 P_{k+1} 中不在同一组中。
(2) 若无法找到 P_k，任意一个输入 x 均可以区分（s_i，s_j），$\eta = x$，算法结束。
(3) 若找到 P_k，CS 的长度为 $k+1$。CS 的初始值 $\eta = \varepsilon$。从 $l=k$ 开始，循环执行，直到 $l=1$：
  ① 在 P_l 表中寻找一个输入 x，x 可以区分状态（s_i，s_j）。即在 P_l 表中，（s_i，s_j）在输入 x 的作用下，其后继状态的第二个下标不同。若有多个 x 可以区分状态（s_i，s_j），则任意选择其中的一个作为输入 x。将 x 添加到 η，即 $\eta = \eta x$。
  ② 在输入 x 的作用下，状态（s_i，s_j）分别转换到（s_i'，s_j'），$s_i = s_i'$，$s_j = s_j'$。
  ③ $l = l-1$。
(4) 选择任意一个输入 x 均可以区分（s_i，s_j），即 $\omega(s_i, x) \neq \omega(s_j, x)$。将 x 添加到 η，$\eta = \eta x$。

在特征序列的基础上，构造特征集就比较简单了。令 $W = \varnothing$，根据（s_0，s_1），（s_0，s_2），…，（s_0，s_n），（s_1，s_2），…，（s_{n-1}，s_n）中依次选择一对状态，求出其特征序列 η_k，并将其添加到 W 中，$W = W \cup \{\eta_k\}$。

例 5-13　求如图 5-31 所示的状态机的特征集。

如图 5-31 所示的状态机，若要计算（s_0，s_1）的特征序列。首先设 $\eta = \varepsilon$，根据算法 5-4 所示，找出（s_0，s_1）所对应的 P_k 和 P_{k+1}。在 P_3 中，（s_0，s_1）在同一组中，而 P_4 中，（s_0，s_1）不在同一组中，显然可以得到 $k=3$。

根据 P_3 表，在 P_3 中，输入 a 以后的后继状态都是 s_{01}，而输入字母 b 的后继状态为 s_{34} 和 s_{45}，所以选择输入 b 添加到特征序列中，$\eta = \eta b = b$。（s_0，s_1）在输入 b 的作用下，状态分别转换成为（s_3，s_4）。

根据 P_2 表，输入 a，后继状态为（s_{22}，s_{11}），依据第二个下标，它们不在同一个组内。而输入 b，后继状态为（s_{43}，s_{33}），依据第二个下标，它们在同一个组内。所以选择输入 a 添加到特征序列中，这时 $\eta = \eta a = ba$。（s_3，s_4）在输入 a 的作用下，状态转换成为（s_2，s_1）。

根据 P_1 表，无论输入是 a 还是 b，其后继状态的第二个下标均不相同，后继状态均不在同一个组中，所以可以在 a 和 b 中任意选择一个。现在选择输入 a 添加到特征序列中，$\eta = \eta a = baa$，（s_2，s_1）在输入 a 的作用下，状态转换成为（s_4，s_0）。

由于 $k=3$，（s_0，s_1）为 3 阶等价。在原始状态转换表中，选择一个区分（s_4，s_0）的输入。因为 $\omega(s_4, a) \neq \omega(s_0, a)$，而 $\omega(s_4, b) \neq \omega(s_0, b)$，所以选择最后一个输入 a 添加到特征序列中，$\eta = \eta a = baaa$。

自此，已经求出（s_0，s_1）的特征序列中 $\eta = baaa$。将其添加到特征集中，$W = \{baaa\}$。

同理，可以计算（s_3，s_4）的特征序列。（s_3，s_4）在 P_2 表中在同一个组，而在 P_3 中不在同一个组中，所以 $k=2$，设 $\eta = \varepsilon$。

根据 P_2 表，输入 b 时，其后继状态的（s_{33}，s_{43}）下标相同，而输入 a 时，其后继状态分别为（s_{22}，s_{11}），所以选择 a 作为输入，所以输入序列为 $\eta = \eta a = a$。

根据 P_1 表，无论输入是 a 还是 b，其后继状态的第二个下标均不相同，后继状态均不在同一个组中，所以可以在 a 和 b 中任意选择一个。现在选择输入 a 添加到特征序列中，

$\eta = \eta a = aa$，(s_2, s_1)在输入 a 的作用下，状态转换成为(s_4, s_0)。

在原始状态转换表中，选择一个区分(s_4, s_0)的输入。因为$\omega(s_4, a) \neq \omega(s_0, a)$，而$\omega(s_4, b) \neq \omega(s_0, b)$，所以选择最后一个输入 a 添加到特征序列中，$\eta = \eta a = aaa$。

自此已经求出(s_3, s_4)的特征序列为 aaa，将其添加到特征集中，W={baaa,aaa}。

同理，可以求出(s_0, s_2)、(s_1, s_2)的特征序列分别为 aa，W=W∪{aa}={baaa,aaa,aa}。求出(s_0, s_3)、(s_0, s_4)、(s_1, s_3)、(s_1, s_4)、(s_2, s_3)、(s_2, s_4)的特征序列均为 a，W=W∪{a}={baaa,aaa,aa,a}。在特征集中特征序列作用下，其输出如表 5-23 所示。

表 5-23　图 5-31 状态机特征集及其输出

当前状态	输入 baaa	输入 aaa	输入 aa	输入 a
s_0	1101	000	00	0
s_1	1100	000	00	0
s_2	1000	010	01	0
s_3	1101	101	10	1
s_4	1100	100	10	1

例 5-14　求图 5-39 状态机的特征集。

画出状态机对应的状态转换表，如表 5-24 所示。

表 5-24　和图 5-39 对应的状态转换表

当前状态	输出		后继状态	
	a	b	a	b
s_0	0	0	s_1	s_3
s_1	0	0	s_0	s_1
s_2	1	0	s_3	s_0
s_3	1	0	s_3	s_2

根据转换表，可以分别构建状态等价划分表 P_1 和 P_2 表，如表 5-25 和表 5-26 所示。

表 5-25　和图 5-39 对应的 P1 表

分组	当前状态	后继状态	
		a	b
1	s_0	s_{11}	s_{32}
	s_1	s_{01}	s_{11}
2	s_2	s_{32}	s_{01}
	s_3	s_{32}	s_{22}

表 5-26　和图 5-39 对应的 P2 表

分组	当前状态	后继状态	
		a	b
1	s_0	s_{12}	s_{34}
2	s_1	s_{01}	s_{12}
3	s_2	s_{34}	s_{01}
4	s_3	s_{34}	s_{23}

求(s_0, s_1)状态的特征序列。(s_0, s_1)在表 P_1 中是同一组，而在 P_2 中不在同一组内。所以 $k=1$。根据 P_1 表，在输入为 a 的情况下，后继状态的第二个下标均为 1，而在 b 输入

条件下，(s_0,s_1)的后继状态分别为(s_{32},s_{11})，所以第一个输入字母为b，状态转换成为(s_3,s_1)。在原始状态转换表中，输入字母a能区分(s_3,s_1)。所以η=ba，W={ba}。同理，可以求出(s_2,s_3)的区分序列也是ba。而(s_0,s_2)、(s_0,s_3)、(s_1,s_2)、(s_1,s_3)由于不存在状态等价，由其原始状态转换表可以知道特征序列均为a。W={a}∪W={a,ba}。不同状态输入特征集的序列以后的输出情况如表5-27所示。

表5-27　图5-31状态机的特征集及其输出

当前状态	输入a	输入ba	当前状态	输入a	输入ba
s_0	0	01	s_2	1	00
s_1	0	00	s_3	1	01

前面已经介绍过迁移覆盖的测试状态集P，结合状态机的特征集，可以构造完整的W方法测试序列。一般而言，W方法的测试序列集由以下几个步骤所构成。

(1) 构造其迁移覆盖集的测试序列P。

(2) 计算M_s的特征集W。

(3) 形成完整的测试集T=P·W。

在具体执行过程中，从M_I的初始状态S_0出发，选择一条迁移覆盖的路径达到状态s_j，执行所有W中的特征序列，每执行完一条特征序列，均需要执行复位Reset操作，使得M_I回到初始状态，如图5-40所示。

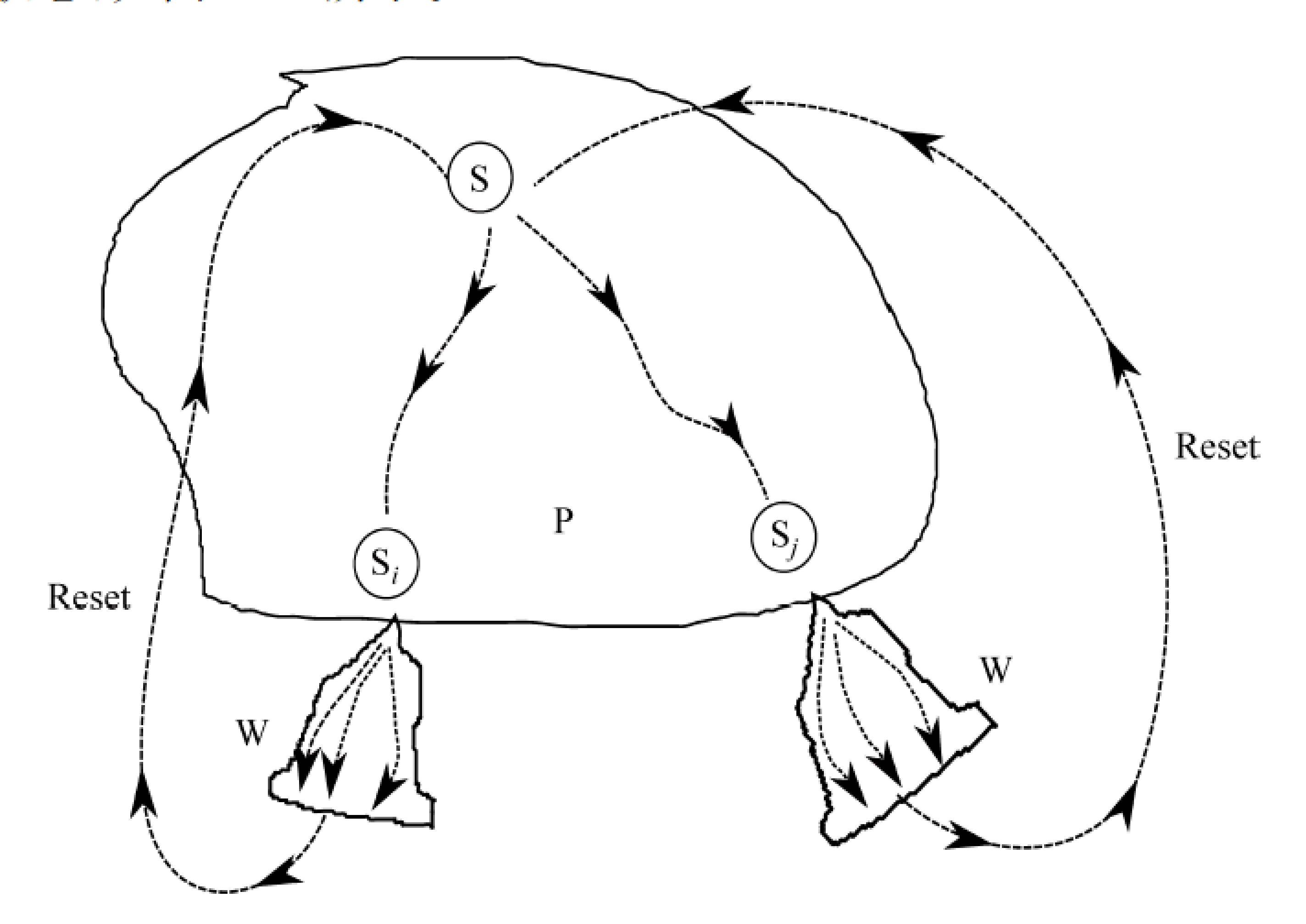


图5-40　W方法执行的过程

如图5-31所示的状态机中，前面已经求得

P={ε,a,b,ba,bb,baa,bab,baaa,baab,baaaa,baaab}

W={baaa,aaa,aa,a}

所以计算得出：

T =P·W

={ε}·W∪{a}·W∪{b}·W∪{ba}·W∪{bb}·W∪{baa}·W∪{bab}

$\cdot W \cup \{baaa\} \cdot W \cup \{baab\} \cdot W \cup \{baaaa\} \cdot W \cup \{baaab\} \cdot W$

$= \{baaa, aaa, aa, a\}$

$\cup \{a \cdot baaa, a \cdot aaa, a \cdot aa, a \cdot a\}$

$\cup \{b \cdot baaa, b \cdot aaa, b \cdot aa, b \cdot a\}$

$\cup \{ba \cdot baaa, ba \cdot aaa, ba \cdot aa, ba \cdot a\}$

$\cup \{bb \cdot baaa, bb \cdot aaa, bb \cdot aa, bb \cdot a\}$

$\cup \{baa \cdot baaa, baa \cdot aaa, baa \cdot aa, baa \cdot a\}$

$\cup \{bab \cdot baaa, bab \cdot aaa, bab \cdot aa, bab \cdot a\}$

$\cup \{baaa \cdot baaa, baaa \cdot aaa, baaa \cdot aa, baaa \cdot a\}$

$\cup \{baab \cdot baaa, baab \cdot aaa, baab \cdot aa, baab \cdot a\}$

$\cup \{baaaa \cdot baaa, baaaa \cdot aaa, baaaa \cdot aa, baaaa \cdot a\}$

$\cup \{baaab \cdot baaa, baaab \cdot aaa, baaab \cdot aa, baaab \cdot a\}$

考虑到最后的复位操作，那么最后执行的测试序列集合为：

$T = \{baaa, raaa, raa, ra,$

$ra \cdot baaa, ra \cdot aaa, ra \cdot aa, ra \cdot a,$

$rb \cdot baaa, rb \cdot aaa, rb \cdot aa, rb \cdot a,$

$rba \cdot baaa, rba \cdot aaa, rba \cdot aa, rba \cdot a,$

$rbb \cdot baaa, rbb \cdot aaa, rbb \cdot aa, rbb \cdot a,$

$rbaa \cdot baaa, rbaa \cdot aaa, rbaa \cdot aa, rbaa \cdot a,$

$rbab \cdot baaa, rbab \cdot aaa, rbab \cdot aa, rbab \cdot a,$

$rbaaa \cdot baaa, rbaaa \cdot aaa, rbaaa \cdot aa, rbaaa \cdot a,$

$rbaab \cdot baaa, rbaab \cdot aaa, rbaab \cdot aa, rbaab \cdot a,$

$rbaaaa \cdot baaa, rbaaaa \cdot aaa, rbaaaa \cdot aa, rbaaaa \cdot a,$

$rbaaab \cdot baaa, rbaaab \cdot aaa, rbaaab \cdot aa, rbaaab \cdot a\}$

在 W 方法中，允许 M_S 和 M_I 的状态不一致，实际上允许 M_I 的状态大于 M_S 状态数量。在这种情况下，利用序列集合 Z 替代特征集 W。

若有两个输入序列的集合 Σ_1 和 Σ_2，定义两个序列集合的连接操作：

$$\Sigma_1 \cdot \Sigma_2 = \{sq_1 \cdot sq_2, sq_1 \in \Sigma_1, sq_2 \in \Sigma_2\}$$

这里，· 表示两个序列连接。在此基础上，定义序列集合的 i 次联合操作：

$$\Sigma^i = \Sigma^{i-1} \cdot \Sigma$$

允许 M_S 的状态为 n，而 M_I 的状态为 m。由于 M_S 是最简的，所以 $m \geqslant n$。定义：

$$Z = (\{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \cdots \cup \Sigma^{m-n}) \cdot W$$

这里，∪是集合的并操作。当 $m > n$ 时将采用 Z 代替 W。

在 $m = n$ 时，$Z = \{\varepsilon\} \cdot W = W$。在具体测试时，每一个序列都是通过 Reset 操作从初始状态开始。为了执行识别一个状态，必须执行特征集中所有输入序列。

例如：$\Sigma = \{a, b\}$，$W = \{a, aa, aaa, baaa\}$，$m = 6$，$n = 5$。那么可以构建 Z 序列如下：

$Z = (\{\varepsilon\} \cup \Sigma^{(6-5)}) \cdot W = W \cup \Sigma. W = \{a, aa, aaa, baaa\} \cup \{a, b\}. \{a, aa, aaa, baaa\}$

$= \{a, aa, aaa, baaa, aa, aaa, aaaa, baaaa, ba, baa, baaa, bbaaa\}$

从上面的分析可以看出，一般使用 W 方法必须满足如下要求。

(1) 对每一个状态的测试都必须使用 Reset 功能，测试序列长度在实际的复杂系统中很难使用。并且要求该功能在 M_I 中已经得到了正确实现。

(2) 要求 M_S 必须是最精简的，否则特征集不存在。这是存在特征集的充要条件。

(3) M_S 和 M_I 所有的状态，应该是从初始状态出发时可达的。

5.4.7 唯一输入/输出序列(U 方法)

U 方法，指采用唯一输入/输出序列进行状态转换来检测状态的方法。一个状态 s 的 UIO 序列是能够唯一标识状态的输入输出序列，对于一个 M_S 其形式定义如下。

对于任意一个状态 s_i，y 是一个输入序列，$y/\omega(s_i, y)$ 是状态 s_i 的一个 UIO 序列，当且仅当：$\forall s_j \in S, y/\omega(s_i, y) \neq y/\omega(s_j, y)(s_i \neq s_j)$。

状态机处在一定的状态时，接收了对应的 UIO 中的输入序列，所产生的输出不同于状态空间中的任何其他状态在接收此输入时所产生的输出。为了方便描述 UIO 序列的产生方法，先定义路径向量。

路径向量(Path Vector，PV)：给定一个状态机 M，路径向量 PV 是状态对的集合 $PV=(s_1/s_1', \cdots, s_i/s_i', \cdots, s_k/s_k')$，$s_1$ 和 s_1' 分别代表了两个路径上的起始状态和当前状态。同一个输入序列将作用在路径向量中的所有路径。

初始状态向量函数(Initial State Vector，ISV)：由 PV 项目中所有的起始状态所构成的状态向量，称为初始状态向量。$ISV(PV)=ISV(s_1/s_1', \cdots, s_i/s_i', \cdots, s_k/s_k')=(s_1, \cdots, s_i, \cdots, s_k)$。

当前状态向量函数(Current State Vector，CSV)：由 PV 项目中所有的当前状态所构成的状态向量，称为当前状态向量。$CSV(PV)=CSV(s_1/s_1', \cdots, s_i/s_i', \cdots, s_k/s_k')=(s_1', \cdots, s_i', \cdots, s_k')$。

转换函数 Tr(PV，a/x)：设路径向量转换 Tr 函数，PV 中的每一个状态对，在输入 a 的作用下转换到新的状态对，其输出为 x，a/x 是状态机的一个变迁，新状态对所构成的向量为 PV′，$PV'=Tr(PV, a/x)=\{s_1/s_1'' \mid s_1''=\delta(s_1', a) \wedge \omega(s_1', a)=x, s_1/s_1' \in PV\}$。

初始路径向量：定义向量$\{s_1/s_1, \cdots, s_i/s_i, \cdots, s_n/s_n\}$为初始向量。初始向量作为推导 UIO 序列的基本出发点。没有任何的路径和初始向量相关联。

从初始路径向量出发，不断应用扩展函数 Tr(PV，a/x)得到新的路径向量。若状态机包含 n 个输入、m 个输出，那么输入和输出的组合共有 $n \times m$ 种可能性。一个路径向量最多可能会产生 $n \times m$ 个新的路径向量。将路径向量看成一个节点，而将输入/输出作为边的标号，那么扩展函数 Tr 就会产生一个节点最多有 $n \times m$ 个孩子的树，这棵树称为 UIO 树。若将初始节点看成树的第一层，在 UIO 树中第一层路径向量 PV 应用扩展函数 Tr 得到所有后继路径节点作为 UIO 树的第二层，以此类推将产生一个多层的 UIO 树，处于同一个层的路径向量处于同一辈。UIO 树的第 k 层最多有 $(n \times m)^k$ 个节点。从理论上，UIO 树可以达到无限层，而无限层的 UIO 树对于解决问题并没有帮助。满足以下三个条件之一的节点，将其标注为终止节点，不再进行扩展。

(1) 存在单实例路径向量(D_1)：若一个路径向量仅包含一个状态对，那么该路径向量被称为单实例模式。$PV=\{s_0/s_1\}$、$PV=\{s_4/s_6\}$都是单实例向量的例子。

(2) 存在同质路径向量(D_2)：若一个路径向量的当前状态向量的成员都相同，则该路径向量称为同质路径向量。

(3) 存在重复路径向量(D_3)：从根节点开始到当前路径向量 PV 的中间节点 PV''，$PV'=Tr(PV,a/x)\subseteq PV''$，那么表明该路径向量要么是在前面已经出现过了，要么是中间某一个路径向量的子集。

通过上面的分析，可以得到构造 UIO 树的算法。从根节点出发，不断进行应用扩展函数进行扩展，若满足上面三个条件，则标记为终止节点，否则做进一步的扩展。直到没有可以扩展的节点为止。具体描述见算法 5-5。

算法 5-5　构造 UIO 序列

(1) 构造初始路径向量 PV，并将其作为 UIO 树的根，并标记为第一层。将初始路径向量 PV 添加到队列中。
(2) 当队列非空时，循环执行：
　① 对于 UIO 的 $i(i\geqslant 1)$层，查找非终止路径向量。
　若找到非终止路径向量，将其移出队列，并应用扩展函数 $T_r(PV,a/x)$计算新一层的 PV''，其中，$a\in\Sigma$。否则，算法终止。
　② 对步骤①中产生的 PV'以及对于输入输出添加到 UIO 树中，判断其是否满足 D_1,D_2,D_3。若满足三个条件中任意一个，则将其标记为终止节点。否则将路径向量添加到块队列中。
(3) 从根节点的路径向量 PV 开始，到达不同起始状态最小路径构成 UIO 序列。

例 5-15　求图 5-41 中有限状态机的 UIO 序列。

在图 5-41 中，输入为 a 和 b，而输出为 0 和 1。所有构成的输入组合共有 a/0，a/1，b/0，b/1 这 4 种情况。初始向量为 $PV_0=(s_0/s_0,s_1/s_1,s_2/s_2)$。将 PV_0 压入队列。

从队列中弹出 PV_0，并应用扩展函数：

$$PV_{11}=Tr(PV_0,a/0)=(s_0/s_2,s_1/s_0)$$
$$PV_{12}=Tr(PV_0,a/1)=(s_2/s_1)$$
$$PV_{13}=Tr(PV_0,b/0)=(s_2/s_0)$$
$$PV_{14}=Tr(PV_0,b/1)=(s_0/s_1,s_1/s_2)$$

图 5-41　求 UIO 的有限状态机

在这 4 个 PV 中，PV_{12} 和 PV_{13} 都是单实例路径向量，将其标记为终止节点。所以在根节点基础上，将 PV_{11} 和 PV_{13} 添加到队列中。第一层处理完毕。

从队列中弹出 PV_{11}，执行扩展函数：

$$PV_{21}=Tr(PV_{11},a/0)=(s_1/s_2)$$
$$PV_{22}=Tr(PV_{11},a/1)=(s_0/s_1)$$

新产生的 PV_{21} 和 PV_{22} 都是单实例，接着 PV_{13} 出队，并执行扩展函数：

$$PV_{23}=Tr(PV_{11},b/0)=(s_1/s_0)$$
$$PV_{24}=Tr(PV_{11},b/1)=(s_0/s_2)$$

至此，队列为空。

选择从根节点开始，各个状态到单实例向量的最短路径，若有两个序列的长度相同，则任选一个，构成了UIO序列，如图5-42所示。

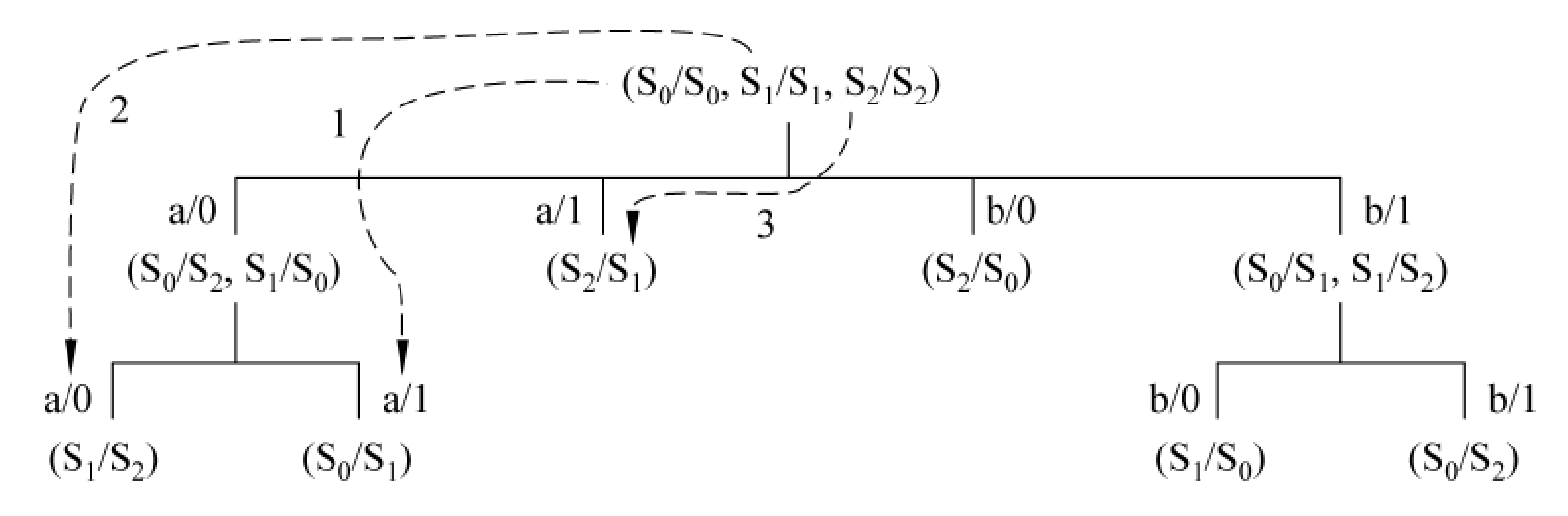


图5-42　依据图5-41产生的UIO树

状态 s_0：{a/0，a/1}。在图5-42中沿着路径1的所有输入/输出对所构成的序列。

状态 s_1：{a/0，a/0}。在图5-42中沿着路径2的所有输入/输出对所构成的序列。

状态 s_2：{a/1}。在图5-42中沿着路径3的所有输入/输出对所构成的序列。

例5-16　构建图5-39的UIO序列。

在图5-39中，其输入/输出的组合，共有三种：a/0，a/1，b/0。在构建UIO树的过程中，每一个节点最多有三个孩子。首先构建初始路径向量 $PV_0=(s_0/s_0, s_1/s_1, s_2/s_2, s_3/s_3)$。将 PV_0 进入队列。

从队列中弹出 PV_0，并应用扩展函数，得到：

$$PV_{11}=Tr(PV_0, a/0)=(s_0/s_1, s_1/s_0)$$
$$PV_{12}=Tr(PV_0, a/1)=(s_2/s_3, s_3/s_3)$$
$$PV_{13}=Tr(PV_0, b/0)=(s_0/s_3, s_1/s_1, s_2/s_0, s_3/s_2)$$

PV_{11}、PV_{12} 和 PV_{13} 构成UIO树的第一层，如图5-43(a)所示。其中，PV_{12} 的当前状态均是 s_3，所以 PV_{12} 是同质路径向量，不再进行扩展。对于 PV_{11} 和 PV_{13} 分别进行进一步扩展。下面以 PV_{11} 为例，做进一步讨论，对于 PV_{11}：

$$PV_{21}=Tr(PV_{11}, a/0)=(s_0/s_0, s_1/s_1)$$
$$PV_{22}=Tr(PV_{11}, b/0)=(s_0/s_1, s_1/s_2)$$

对于 PV_{11} 中的两个当前状态 s_1 和 s_0，不存在满足a/1输入条件的。而 PV_{21} 的路径向量是初始路径向量的子集，所以不再进行扩展。对于 PV_{22}：

$$PV_{31}=Tr(PV_{22}, a/0)=(s_0/s_0)$$
$$PV_{32}=Tr(PV_{22}, a/1)=(s_1/s_3)$$
$$PV_{33}=Tr(PV_{22}, b/0)=(s_0/s_1, s_1/s_2)$$

PV_{31} 和 PV_{32} 都是单实例路径向量，不再扩展。而 PV_{33} 分别对输入输出做进一步的扩展：

$$PV_{31}=Tr(PV_{33}, a/0)=(s_0/s_0)$$
$$PV_{32}=Tr(PV_{33}, a/1)=(s_1/s_3)$$
$$PV_{33}=Tr(PV_{33}, b/0)=(s_0/s_1, s_1/s_0)$$

这里，三个路径向量均是终止节点，不再扩展，如图5-43(b)所示。

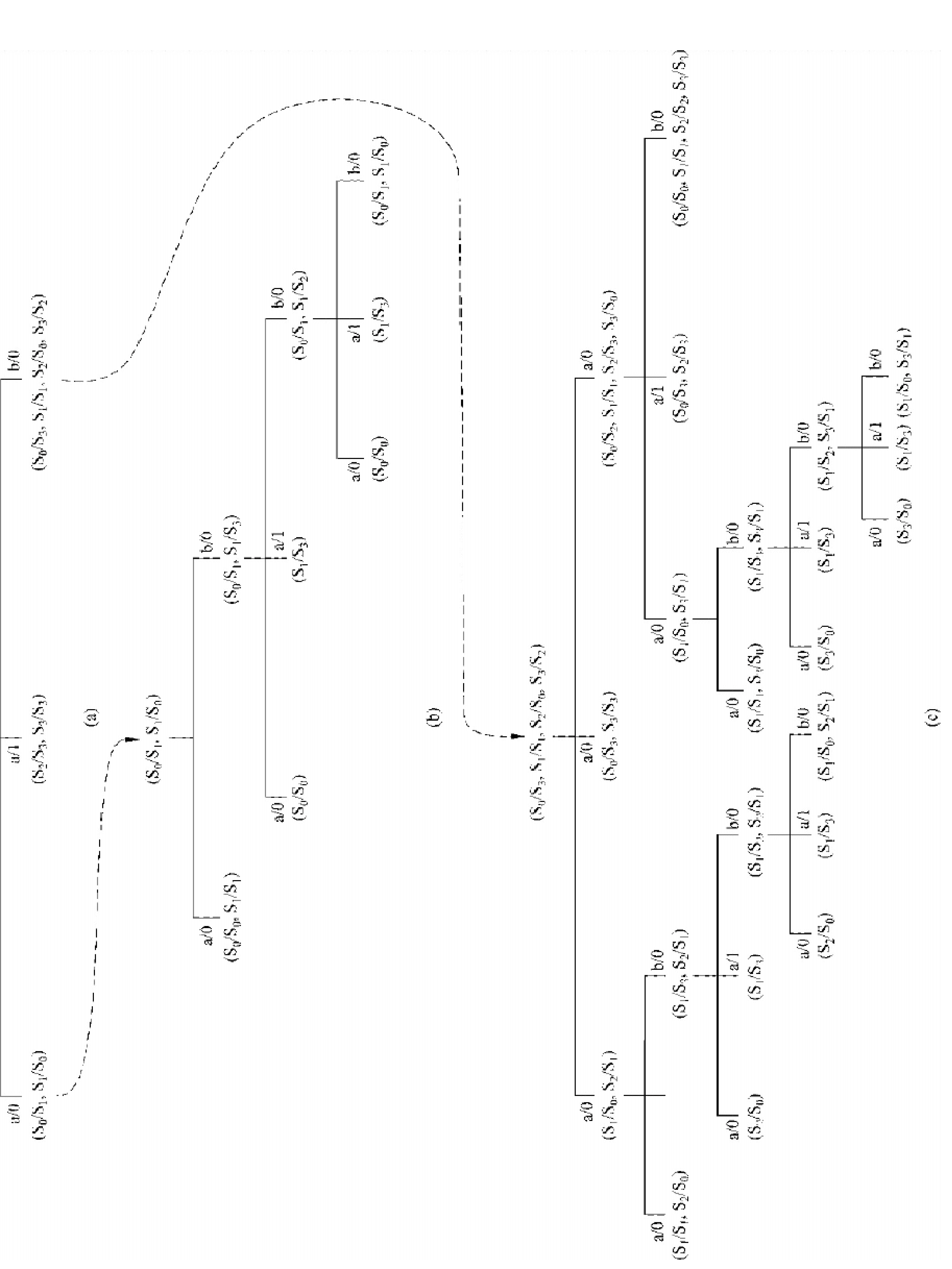


图 5-43 依据图 5-39 产生的 UIO 树

接着对 PV_{13} 做进一步扩展，如图 5-43(c)所示，最后形成一棵完整的 UIO 树。

在生成的 UIO 图中，对于初始路径向量中的每一个状态，从根节点开始，沿着 UIO 寻找一条最短的路径到其对应的单实例路径向量的叶子节点。构成所有的 UIO 序列。

状态 s_0：{a/0，b/0，a/0}

状态 s_1：{a/0，b/0，a/1}

状态 s_2：{b/0，a/0，b/0，a/0}

状态 s_3：{b/0，b/0，a/0，b/0，a/0}

在 UIO 测试方法中，每一个状态对应一条 UIO 序列。对于采用变迁覆盖到达的状态，将采用 UIO 序列进行校验。D 方法需要执行特征集中的所有序列才能区分(或者说标识)一个状态，和 D 方法不同，仅需要和该状态对应的 UIO 序列便可以标识一个状态。在 U 方法中，在变迁覆盖的基础上，需要执行 UIO 序列，如图 5-44 所示，包括以下三个步骤。

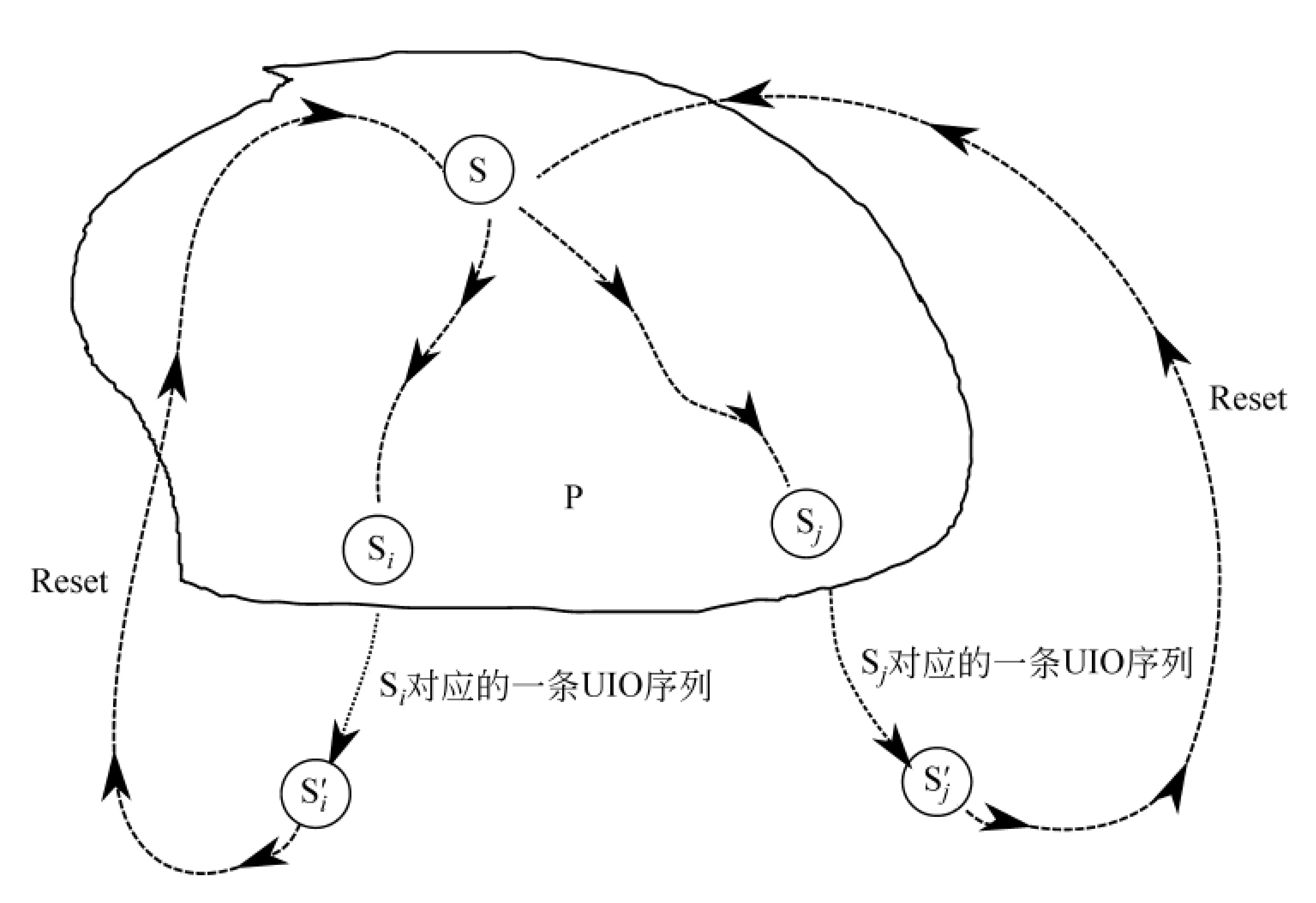


图 5-44　UIO 方法的原理

(1) 执行变迁覆盖的测试序列，到达某一状态 s_i。

(2) 执行 s_i 对应的 UIO 序列，判断其状态是否正确。

(3) 执行复位操作，使其恢复初始状态。

例 5-17　求如图 5-41 所示的状态机的 UIO 测试序列。

如图 5-41 所示的状态机，可以求出其迁移覆盖集合。

P＝{ε，a，b，aa，ab，ba，bb}，将迁移覆盖的输入字符串和其所到达的状态的 UIO 串的输入字符连接起来，构成了 UIO 方法的最终输入字符串，如表 5-28 所示。

例 5-18　求如图 5-41 所示的状态机的 UIO 测试序列。

如图 5-41 所示的状态机，可以求出其迁移覆盖集合。

P＝{ε，a，b，aa，ab，ba，bb}，将迁移覆盖的输入字符串和其所到达的状态的 UIO 串的输入字符连接起来，构成了 UIO 方法的最终输入字符串，如表 5-29 所示。

表 5-28 图 5-41 状态机的 UIO 测试序列

覆盖变迁	预期状态	变迁覆盖输入序列	UIO 序列	最终测试序列 • 表示字符串连接
—	s_0	ε	a/0,a/1	I:r • aa O: -01
s_0-s_2	s_2	a	a/1	I:r • a O: -0 • 1
s_0-s_1	s_1	b	a/0,a/0	I:rb • aa` O: -1 • 00
s_0-s_2-s_1	s_1	aa	a/0,a/0	I:raa • aa O: -01 • 00
s_0-s_2-s_0	s_0	ab	a/0,a/1	I:rab • aa O: -00 • 01
s_0-s_1-s_0	s_0	ba	a/0,a/1	I:rba • aa O: -10 • 01
s_0-s_1-s_2	s_2	bb	a/1	I:rbb • a O: -11 • 1

表 5-29 图 5-41 状态机 UIO 测试序列

覆盖变迁	预期状态	变迁覆盖输入序列	UIO 序列	最终测试序列 • 表示字符串连接
-	s_0	ε	a/0,b/0,a/0	I: r • aba O: - • 000
s_0-s_1	s_1	a	a/0,b/0,a/1	I: ra • aba O: -0 • 001
s_0-s_3	s_3	b	b/0,b/0,a/0,b/0,a/0	I:rb • bbaba O: -0 • 00000
s_0-s_1-s_0	s_0	aa	a/0,b/0,a/0	I:raa • aba O: -00 • 000
s_0-s_1-s_1	s_1	ab	a/0,b/0,a/1	I:rab • aba O: -00 • 001
s_0-s_3-s_3	s_3	ba	b/0,b/0,a/0,b/0,a/0	I:rba • bbaba O: -01 • 00000
s_0-s_3-s_2	s_2	bb	b/0,a/0,b/0,a/0	I:rbb • baba O: -00 • 0000
s_0-s_3-s_2-s_3	s_3	bba	b/0,b/0,a/0,b/0,a/0	I:rbba • bbaba O: -001 • 00000
s_0-s_3-s_2-s_0	s_0	bbb	a/0,b/0,a/0	I:rbbb • aba O: -000 • 000

值得注意的是，并非所有的状态机都具有 UIO 序列。在图 5-45 中，在输入 a 时，s_0 和 s_2 输出 0，状态转移到 s_1，以 a 字符无法区分 s_0 和 s_2。在输入 b 时，s_0 和 s_1 输出 0，状态转移到 s_2，以 b 字符无法区分 s_0 和 s_1。这两种情况表述的路径向量如图 5-46 所示，不存在以 s_0 为起始状态的单实例路径向量。因此状态 s_0 不存在 UIO。

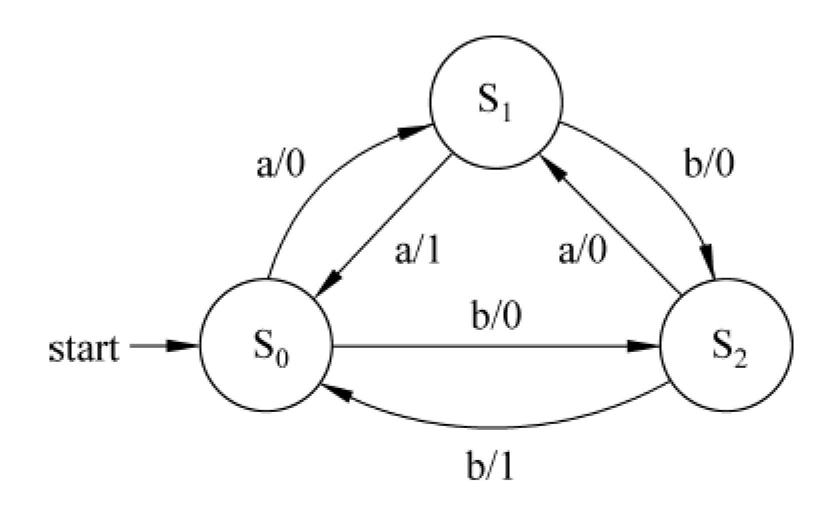


图 5-45　状态 S0 不存在 UIO 序列

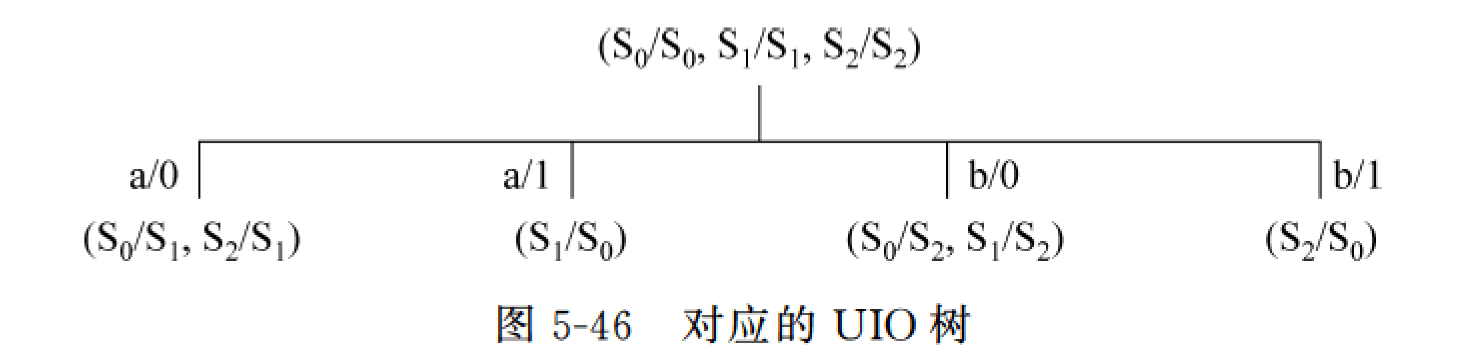


图 5-46　对应的 UIO 树

第 6 章　面向对象结构的软件测试

施乐公司于 20 世纪 80 年代在系统设计中引入对象、对象类、方法、实例。基于以往已提出的有关信息隐蔽和抽象数据类型等概念，逐步发展和建立起较完整的 OO 方法的概念理论体系和实用的软件系统。本章以 Python 语言为例，详细介绍面向对象结构的软件测试方法，包括对象属性、类属性、对象的创建和销毁、装饰器、多态等主题。

6.1　Python 面向对象

在软件危机之前，软件需求主要是数值计算问题，并且数值处理的需求也相对简单，面向过程的软件方法能够精确描述这种需求。面向过程是一种以过程为中心的编程思想，将需要解决的问题划分为一个个过程。在这个面向过程分析中，蕴含一个假设，这个过程是稳定、不变的，并且过程中每一个步骤都是固定的，有非常严谨的耦合关系。面向过程的方法在描述局部化的细节时具有明显的优势，将世界看成一个过程化的、紧密相连的小系统，不同系统之间存在非常紧密的耦合关系。到 20 世纪 60 年代，随着计算机应用范围的扩展，需求越来越复杂，功能也越来越多。面向过程在宏观上描述复杂系统之间的逻辑关系时显得力不从心，同时当软件需求发生变化时，导致软件维护编程非常困难，软件可靠性问题也越来越突出。

面向过程的软件开发主要存在以下几个问题。

(1) 软件的可重用性差。重用性是指事物不经修改或者稍加修改就可以重复使用的性质。在面向过程的软件开发中，重用的主要途径就是函数调用。然而在一个系统中，存在多个类似但不相同的，或者存在部分相同的功能，通过函数调用复用就显得非常牵强。

(2) 软件可维护性差。增强软件可维护性是发展面向对象编程的核心需求，面向过程系统中形成了大量的紧耦合的关系，使得子系统的边界模糊，功能划分困难，导致维护成本大量上升。

(3) 软件无法满足用户的需求变化。在面向过程的开发方法中，通过自顶向下方式，将功能不断分解，然后实现最底层的处理需求，在整个需求的表达过程中以“过程”来构造系统。当用户需求的变化以后，可能导致整个分解过程失效。

在这种背景下，无论是产业界还是学术界都开始研究面向对象的开发方法。在 20 世纪 90 年代，面向对象成为主流的软件开发方法。面向对象的核心思想认为世界是由许多具有一定功能和属性的对象所构成，不同对象之间的相互作用和通信组成了各种现实系统。信息世界依据现实世界中的对象来构造系统，而不是围绕总功能来分解功能。

面向对象方法将世界看成一个个对象，对象和对象通过消息进行交互并传递信息。例如，在一个航班管理系统中，涉及航班、飞机、飞行员。对象之间相对独立，在没有和其他对象通信时，各个对象是没有关联的。一般而言，一个对象通过消息和有限对象进行交

流。每一个对象都具有属性和方法,属性是指对象所具有的性质(数据值),方法是对象所具有的一种功能。例如,飞行员A,他的姓名name、飞行里程mileage、年龄age都是他的属性,驾驶飞机的动作fly()是飞行员A的方法。一组有相同数据和相同操作的对象称为类,类是对象的模板。类是对象之上的抽象,对象是类的具体化。

面向对象具有三个重要的特性:封装、继承、多态。对象的功能以及属性,除非必要,否则对其他对象是不可见的。从其他对象看,一个对象除了必要的消息通信以外,对象内部机制是不可见的,这个特性称为封装。封装特性能够使需要考虑的问题局部化。封装使数据和加工该数据的方法(函数)封装为一个整体,以实现独立性很强的模块,使得用户只能见到对象的外特性(对象能接收哪些消息,具有哪些处理能力),而对象的内特性(保存内部状态的私有数据和实现加工能力的算法)对用户是隐蔽的。和生物体一样,对象可以产生子辈,子辈具有父辈的全部功能,在需要时子辈也会发展父辈的功能,称为继承。继承支持系统的可重用性,从而达到减少代码量的作用,而且还促进系统的可扩充性。多个对象具有相同的界面,但是其实现的行为却不相同,该特性称为多态。这些特性为描述和解决现实世界中的复杂问题提供了有力的手段。

面向对象的核心概念是类和对象。

类是具有相同和类似性质的对象的抽象。类具有属性和操作。

类拥有属性,属性是对象状态的抽象,一般用数据来表示类的属性。类的属性可以是简单数据类型、组合数据类型,也可以是一个类。属性可以理解为特殊的变量,因为这个变量是属于这个类的,类的每一个实例也都拥有它。Python中的类属性和Java语言中的静态属性类似。

类拥有方法,操作是对象行为的抽象,用一个方法名称或者名称连同实现该方法的代码块来描述。在具有抽象类或者允许将类定义和实现分离的语言中,可以在类中仅有方法名称。从单个操作上看,和面向过程中的函数比较类似,实际上方法就是和特定类绑定的函数。

对象可以是任何事物,从最简单的整数到复杂的飞机等均可看作对象,它可以是具体的事物,也可以是抽象的规则、计划或事件。

对象具有状态,一个对象用数据值来描述它的状态。

对象具有操作,操作是对象的行为,用于改变对象的状态。对象实现了数据和操作的结合,使数据和操作封装于对象的统一体中。操作是类中方法的具体执行。

6.2 Python面向对象编程基础

在面向对象测试的讨论过程中,均以Python作为例子来说明具体的思想。在讨论具体的面向对象测试之前,先简单介绍Python面向对象的编程基础知识。

1. 类定义

在Python中,类的定义格式如下:

```
class className:
```

```
    block
```

这里，class 是 Python 的关键字，className 是类名，类名后面具有冒号。

程序 6-1 给出了一个最简单的类。

程序 6-1　一个最简单的类

```
class Employee:
    pass

p= Employee()
```

前两条语句定义了一个 Employee 类。Python 语言中，pass 关键字作为一个预留位置而存在，表示什么都不执行。这里尽管只有两条语句，但它确实是一个类。语句 p=Employee()给类创建了一个对象。

现在可以给类增加一些方法，和大部分编程语言教程一样，以 Hello 作为介绍类定义。程序 6-2 给出了一个简单的类程序，在这个类中，定义了一个方法 sayHello()。

程序 6-2　具有一个简单方法的类

```
class Employee:
    def sayHello(self):
        print 'hello'
p=Employee()
p.sayHello()
```

在这个例子中，前三行定义了一个类，这类包含一个方法 sayHello()。sayHello()方法中的参数 self 表示该方法和一个具体的对象相联系。

后两行演示对象的创建和调用。通过 p=Employee()，类创建了一个实例（一个具体的对象）。而 p. sayHello()执行了对象 p 的 sayHello()方法的调用。而在调用时，该方法直接依附于对象 p，在调用时不写 self 参数。

2. 对象的创建和初始化

在定义好一个类以后，必须执行创建和初始化，一个对象才开始它的生存周期。在定义类时，__init __特殊方法中，提供了类初始化的机会。类的创建通过类名实现。例如，程序 6-3 定义的类中，通过如下语句创建对象。

```
p1=Employee('Zhangsan', 25)     创建一个姓名为'Zhangsan',年龄为 25 的雇员 p1
p2=Employee('Wangwu', 28)       创建一个姓名为'Wangwu',年龄为 28 的雇员 p2
```

显然 p1 和 p2 为两个不同的对象，它们具有不同的属性。

3. 对象的销毁

Python 销毁对象时，调用__del __()方法。Python 中所有的变量都是指向一个特定对象的应用，只有最后一个引用也消失时，__del __()函数才会被调用。程序 6-3 给出了一个对象销毁的例子。在通过 p1=Employee('clz', 42)创建了一个对象以后，两个引用 p2 和

p3 都指向了该对象,然后调用 del 删除引用。一直到第三个引用被删除以后,__del __()方法才被调用。

程序 6-3　对象的销毁

```
#coding=utf-8
class Employee:
    def __init__(self,name,age):
        self.name=name
        self.__age=age

    def __del__(self):
        print "destroyed"

p1=Employee('clz', 42)
p2=p1
p3=p1
del p1
del p2
del p3
```

4. 类的继承

类的继承的出现使得程序语言能够表现复杂的类层次结构,也可方便复用已经定义过的类。同样,通过类的继承来表述不同类之间的关系。有一个类 A,若通过继承产生了类 B 和类 C,那么 B 和 C 是 A 的子类,而 A 是类 B 和类 C 的父类(超类、基类)。子类会从其父类中继承属性,子类也会继承其父类中所定义的所有属性名称。子类的定义和父类没有区别,只需要在类名后的括号中指定它的父类。其定义结构如下:

```
class 类名 (父类名 1,父类名 2,…):
```

例如,类 Point 继承了类 Graphic,那么可以定义如下:

```
class Point(Graphic):
```

Python 和其他语言不同,子类不会自动调用父类的初始化方法__init __(),必须通过显式调用的方法。程序 6-4 首先定义了一个类 Graphic,然后定义了类 Point 和类 Circle,在 Graphic 中定义了基本坐标属性 x 和 y,在 Point 和 Circle 类中自动继承了这两个属性。在类 Point 的初始化方法__init __()方法中第一条语句就显式调用了父类的方法:

```
Graphic.__init__(self,x,y)
```

程序 6-4　一个简单的类继承示例

```
class Graphic():
    def __init__(self,x,y):
        self.x=x
        self.y=y;
```

```
    def show(self,x,y):
        print "Graphic:",self.x,self.y

class Point(Graphic):
    def __init__(self,x,y,color):
        Graphic.__init__(self, x, y)
        self.color=color
    def show(self):
        print "Point:", self.x,self.y

class Circle(Graphic):
    def __init__(self,x,y,r):
        Graphic.__init__(self,x,y)
        self.r=r
    def show(self):
        print "Circle:", self.x,self.y,self.r

p=Point(4,5,'red')
p.show()
c=Circle(6,6,3)
c.show()
```

6.3 基于类属性和对象属性的测试

在 Python 中,从作用域上来看属性分为类属性和对象属性,从可访问性上看包括公共属性和私有属性。类属性的作用域是类相关的,无论对象是否存在或者对象的多少,类属性都存在。而对象属性依附于一个具体的对象,对象属性的生存周期一定在对象的生存周期内,对象属性的生存周期可能小于类属性的生存周期,但一定不会大于对象的生存周期。

1. 类属性

例如,雇员类,其包含一个所有雇员的共同信息:全公司的雇员数量,雇员类的定义如程序 6-5 所示。为了清晰描述类的属性,在这个例子中,没有定义任何的方法。

程序 6-5 在类外部访问类属性

```
#coding=utf-8
class Employee:
    number=1

print Employee.number
Employee.number=Employee.number+1
print Employee.number
```

这个程序中，前两行定义了一个类。类名为 Employee，这个类具有一个属性 number，这个属性和对象无关。

类属性可以在类的外部通过类名直接访问，其基本格式为：

类名.属性名称

例如，print Employee. number 显示了 Employee 中的类属性的 number。类属性可以直接参加各种运算。Employee. number＝Employee. number＋1 实现几个公司中的雇员数量加 1。

若不通过类名直接访问类属性，将会提示变量没有定义，如下所示。

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>class Employee:
    number=1

>>>print Employee.number
1
>>>print number
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in<module>
    print number
NameError: name 'number' is not defined
>>>
```

即使在类内部访问一个类属性，也同样通过类名进行访问。现在为类增加一个方法 showNumber()，显示 number 属性值，如程序 6-6 所示。

程序 6-6　内部方法访问类属性

```
#coding=utf-8
class Employee:
    number=1
    def showNumber(self):
        print Employee.number;

p=Employee()
p.showNumber()
```

尽管 showNumber()是类的一个方法，但是在访问类属性 number 时，必须通过类名访问。这和 Java、C++ 访问类属性有很大的差别。

2. 对象属性

若一个属性和具体的对象相关，这种属性就是对象属性。对象属性是在对象被创建以后赋予类的一些属性，Python 作为一个动态语言，对象属性可以动态添加。

对象属性的访问：

对象.属性

当给对象的属性赋值时，若指定的对象属性不存在，Python 自动为对象添加属性。若指定的对象属性已经存在，则直接修改属性值。

程序 6-7 给出了一个动态添加对象属性的例子。

程序 6-7　对象属性的动态添加

```
class Employee:
    def sayHello(self):
        print 'hello'

p=Employee()
p.age=5
print p.age
p.name='clz'
print p.name
```

在这个例子中，前三行定义了一个类 Employee。p＝Employee()创建类 Employee 的一个对象。然后，给对象 p 添加了两个属性 name 和 age。p. age＝5 语句为对象 p 增加一个对象属性 age。p. name＝'clz'给对象 p 增加一个对象属性 name。对象属性的添加并不需要做任何的特别声明。

print p. age 给出了访问对象属性的基本方法。

对象属性可以参与一些必要的运算，例如：

```
p.age=p.age+5
```

对象的属性也可以直接删除，若已经添加了属性，则可以使用 del 来删除属性。

```
del p.name
```

除此之外，Python 提供了如下一系列函数进行属性的管理。

getattr(obj, name[, default])：访问对象的属性。

hasattr(obj,name)：检查是否存在一个属性。

setattr(obj,name,value)：设置一个属性。若不存在，创建一个新属性。

delattr(obj, name)：删除属性。

例如：

```
hasattr(p, 'age')          #如果存在 'age' 属性返回 True
getattr(p, 'age')          #返回 'age' 属性的值
setattr(p, 'age', 8)       #添加属性 'age' 值为 8
delattr(p, 'age')          #删除属性 'age'
```

在创建类时，Python 通过类的__init__()方法实现一些初始化动作，在初始化的过程中，通常对一些对象属性初始化。程序 6-8 给出了对象属性和初始化方法的

Employee 类。

程序 6-8　含有对象属性和初始化方法的 Employee 类

```
#coding=utf-8
class Employee:
    companyName="ssc"
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def showName(self):
        print self.name

    def showAge(self):
        print self.age

    def showCompanyName(self):
        print Employee.companyName

p=Employee('clz',42)
p.showName()
p.showAge()
p.showCompanyName()
```

在这个类中，增加了两个对象属性姓名 name 和年龄 age 和一个特殊的方法__init__()方法。在大部分语言中，对象的创建和初始化统称为构造函数，而在 Python 中实例的创建和初始分为两个过程，__new__()用于实例的创建，而__init__()方法是在对象被创建以后进行初始化。在__init__()方法中，两个对象的属性被初始化：self.name=name 和 self.age=age。这里 self 代表的是一个具体的对象，和 Java 语言中的 this 类似。同时在这个类的定义中，增加了两个方法 showName()和 showAge()。

3. 私有属性

和大部分其他语言不同，Python 中的类没有 public、private、protect 等关键词来修饰不同的属性。在 Python 中，默认的属性都是公共属性，如果需要表示私有属性，那么在属性和方法之前增加两个下划线表示。Python 不允许直接访问私有变量。程序 6-9 给出了一个具有私有属性的类的例子。

程序 6-9　具有私有属性的类

```
#coding=utf-8
class Employee:
    def __init__(self,name,age):
        self.name=name
        self.__age=age
```

```
    def showAge(self):
        print self.__age

p=Employee('clz', 41)
p.showAge()
```

类 Employee 具有两个属性：name 和__age。其中，__age 就是私有属性，类内部的方法可以直接访问私有属性。方法 showAge()中的语句 print self. __age 就是这种形式。但是在类的外部不能直接访问，在定义该类以后，增加语句：

```
print p.__age
```

在执行到该语句以后将提示变量不存在：

```
AttributeError: Employee instance has no attribute '__age'
```

实际上，Python 并没有完全阻止对私有属性的访问，对于以__开始的变量，系统自动在私有属性前增加了字段_类名。程序 6-10 中，最后一条语句 p. _Employee __age 表示访问类中的私有变量__age。

程序 6-10　私有属性的外部访问

```
#coding=utf-8
class Employee:
    def __init__(self,name,age):
        self.name=name
        self.__age=age

    def showAge(self):
        print self.__age

p=Employee('clz', 42)
print p._Employee__age
```

4. 基于属性的测试

故障模型 1：同名误用。

在对象没有定义和类同名的属性时，Python 允许通过对象访问类属性。若通过赋值语句没法去更改类属性的值，此时 Python 会为对象创建一个独立的类属性，而屏蔽了对象属性。此时仅从访问语句本身无法判定该语句访问的是类属性还是对象属性，导致属性值的误用。

程序 6-11　同名的类属性和对象属性

```
class C():
    value=100
```

```
c=C()

def test():
    print "c.value:",c.value,"   Id: ",id(c.value)
    print "C.value:",C.value,"   Id: ",id(C.value)
    print

test()
c.value=c.value+1
test()

del c.value
test()
```

这个程序的执行结果如下。

```
c.value: 100    Id: 4894700
C.value: 100    Id: 4894700

c.value: 101    Id: 4894688
C.value: 100    Id: 4894700

c.value: 100    Id: 4894700
C.value: 100    Id: 4894700
```

在这段程序中，在改变 c. value 值之前，c. value 和 C. value 之前的值都是同一个对象，所以前两个 print 语句打印的结果都是一样的，也可以通过下面两条语句验证。

```
print id(c.value)
print id(C.value)
```

特别注意其中的语句：

```
c.value=c.value+1
```

这个语句中等号右边的 c. value 没有值改变过，就是 c. value，而左边 c. value 的值由于与 c. value 的值不一样，Python 将其认为是一个对象属性而不是类属性，也就是这时候添加了对象属性。在中间的两个 print 语句中，显示两个不同的值。

而 del c. value 删除了对象属性，但是类属性仍然存在。在没有对象属性时，直接采用对象属性，所以两者的值是相同的。从上面看出同名导致的误用几率是很大的，在实际工作中，要尽量避免属性同名。凡是存在类属性的类中，必须设计针对类属性的测试用例用于检测其变化。

故障模型 2：类属性的简单对象和可变对象访问误用。

Python 函数参数传递采用“传对象引用”的方式。该方式相当于传统语言传值和传引用的一种综合。如果函数收到的是一个可变对象(比如字典或者列表)的引用，就能修

改对象的原始值——相当于通过“传引用”来传递对象。如果函数收到的是一个不可变对象(比如数字、字符或者元组)的引用,就不能直接修改原始对象——相当于通过“传值”来传递对象。

在类中,对象对类属性的访问和参数访问的情况非常类似。如果对象属性是不可变对象的引用,那么可以通过该引用直接修改类属性的值。

程序 6-12　对象通过可变对象引用修改类属性的值

```
class A:
    cls_i=0
    cls_j=['sofware','testing']

b=A()

def test():
    print b.cls_i
    print A.cls_i
    print b.cls_j
    print A.cls_j
    print

test()

b.cls_i=1
b.cls_j.append('python')
test()

del b.cls_i
b.cls_j.pop(1)
test()
```

这个程序的执行结果如下。

```
0
0
['sofware', 'testing']
['sofware', 'testing']

1
0
['sofware', 'testing', 'python']
['sofware', 'testing', 'python']

0
0
```

```
['sofware', 'python']
['sofware', 'python']
```

在这个例子中，类 A 拥有两个类属性，其中 clz_j 是一个可变对象的引用。b 是 A 类的一个实例，实例 b 直接拥有类 A 的两个属性。对 clz_j 增加一个元素，但是由于并没有改变 b 对应的类属性的引用，所以还是类属性。但是对于 cls_i 值的修改，直接产生了一个新的对象属性，第二个 test() 函数的运行结果能够反映出该结果。同理，通过列表的 pop 方法，对象可以修改类属性的值。

6.4 基于对象创建和销毁的测试

6.4.1 基于类创建和继承测试

在 Python 的类中，存在两个和对象创建相关的内置函数：__init__()和__new__()。但一个类被实例化时，首先调用的方法是__new__()方法，然后是__init__()方法。__new__()用于执行对象的生成，而__init__()方法用于对类执行初始化。

1. 基于类创建的测试

只有在特殊情况下，需要调用__new__()方法，一般情况下不需要单独调用__new__()方法。例如，继承不可变的 class，为控制实例化过程提供精细的控制，或者实现单实例模式。而__init__()是在类实例创建之后调用。例如，需要正数的整数类型时，通过定义类 PositiveInt 继承 int 而得到，而 int 是不可变的类型，必须通过__new__()方法而得到，如程序 6-13 所示。在该例子中定义了__new__()方法，该方法简单调用了父类的__new__()方法，并将 value 的绝对值作为参数传入。

程序 6-13 利用__new__()方法继承不可变类

```
class PositiveInt (int):
    def __new__(cls, value):
        return super(PositiveInt, cls).__new__(cls, abs(value))


i=PositiveInt (-3)
print i
```

单实例的模式具有非常广泛的应用场景。例如，网站的计数器一般采用单例模式实现，否则难以同步。例如，在整个系统运行过程中，回收站一直维护着仅有的一个实例。由于共享的日志文件一直处于打开状态，只能由一个实例去操作，应用程序的日志应用一般采用单例模式实现。

程序 6-14 给出了一个单实例模式的具体例子，先判断类是否已经具有实例，若有，简单地返回该实例作为返回值。如果该类没有实例存在，则调用其父类的__new__()方法。

程序 6-14 利用__new__()方法实现单实例模式

```
class Singleton(object):
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance= super(Singleton, cls).__new__(cls)
        return cls.instance
```

单实例模式，主要解决用户具有多个实例可能出现性能消耗和资源控制混乱的情况。单实例模式，一般通过设计多次调用的类的实例化，并检查是否仅产生一个实例。例如，在程序 6-14 中定义的类，通过下面的语句实现单实例模式的测试。

```
obj1= Singleton()
obj2= Singleton()
obj1.attr1= 'value1'
print obj1.attr1, obj2.attr1
print obj1 is obj2
```

其执行的结果为：

```
value1 value1
True
```

2. 子类隐式继承父类构造方法的测试

继承是面向对象的重要特征之一。继承是两个类或者多个类之间的父子关系，子类继承了父类的所有公有实例变量和方法。继承实现了代码的重用。重用已经存在的数据和行为，可减少代码的重新编写，增加代码的可维护性。

最简单的继承类的方式是：在定义子类时增加括号和父类名称。这种方法并没有声明该类的构造器，Python 会自动隐性调用基类的构造器。

基本语法是：

```
class SubClass(ParentClass):
    class_suite
```

其中，SubClass 是子类的名称，ParentClass 是父类(基类)的名称，class_suite 是类的定义体。如果在 class_suite 中，没有定义__init__()方法，那么会隐式继承父类构造方法。此时的错误模式和测试要点如下。

错误模式：由于在子类的声明中并没有显式定义参数，在进行类实例化时，使用父类的实例化方法，在使用的时候容易遗忘基类中需要的参数。

测试要点：在实例化时检查父类的初始化方法的参数规则。

程序 6-15 给出了一个简单继承的例子，类 Apple 继承了类 Fruit。在类 Apple 中没有定义__init__()方法，直接继承了 Fruit 的__init__()方法。

程序 6-15 子类隐藏继承父类构造方法

```
class Fruit:
```

```
    def __init__(self,color):
        self.color=color

class Apple(Fruit):
    def showcolor(self):
        print "%s Apple" %self.color

myapple=Apple('Red')
myapple.showcolor()
```

程序 6-15 中在类 Apple 中并没有定义__init __()方法。但是其在实例化时，实例化语句为 myapple=Apple('Red')，调用的是基类 Fruit 的__init __()方法。

3. 子类遗漏调用父类构造方法的测试

若子类定义了初始化方法__init __()方法，则此时 Python 不会自动执行父类的__init __()方法。此时的错误模式和测试要点如下。

错误模式：遗漏父类__init __()方法的调用。

测试要点：在子类实例化以后，检查所有的属性值，包括子类和父类的定义对象属性和类属性。

程序 6-16 给出了这种错误模式的例子，子类 Apple 继承了父类 Fruit。并且在子类中定义其自身的__init __()函数，但是没有显式调用父类的__init __()方法。而在 Apple 类中仅初始化了其中的 weight 对象属性，color 对象属性的初始化工作在父类中执行。但是由于其并没有调用父类的方法。在子类中如果调用了该对象属性将导致程序出错。

程序 6-16　子类遗漏调用父类构造方法

```
class Fruit(object):
    def __init__(self,color):
        self.color=color

class Apple(Fruit):
    def __init__(self, weight,color):
        self.weight=weight
        #Fruit.__init__(self, color)
        #super(Apple,self).__init__(color)
    def showweight(self):
        print "Apple's weight is % s" %self.weight
    def showcolor(self):
        print "Apple's color is %s" % self.color

myapple=Apple(22,'Red')
myapple.showweight()
#myapple.showcolor()
```

在 Python 中，调用父类的方法有两种，在早期的版本中通过父类的类名实现调用，另外一种是通过 super 函数进行调用。super 只能应用于新类，而不能应用于经典类。super 方法实现继承的优点是可以不用直接引用基类的名称就可以调用基类的方法。如果改变了基类的名称，那么所有子类的调用将不用改变。所谓经典类就是不继承自其他类的类，程序 6-15 中的 Fruit 类就是经典类。程序 6-16 中间的两条带注释语句给出了两个调用父类初始化方法的例子。

```
#Fruit.__init__(self, color)
#super(Apple,self).__init__(color)
```

尽管没有调用父类的初始化方法，myapple. showweight()仅涉及 Apple 类自身引入的对象属性，所以程序的编译和运行都是正常的。若将程序 6-16 中 myapple. showcolor()前面的注释删除，那么 Python 将提示：AttributeError：Apple instance has no attribute 'color'。若将中间的调用父类初始化两个语句的注释去掉其中一句，并同时去掉 myapple. showcolor()语句前的注释，程序一切正常。

6.4.2　基于多重继承初始化方法的测试

Python 允许一个类继承多个父类，在存在多个父类的情况，类的定义如下：

```
class SubClassName (ParentClass1[, ParentClass2, …]):
    class_suite
```

多重继承允许在括号中填写多个父类。

子类继承自多个基类，且子类未定义自身的__init __()方法时，仅调用第一个基类的__init __()方法。

在这种方式中，其故障模式和测试要点如下。

故障模式：没有留意多重继承中父类的安排次序。

测试要点：所有与首个父类初始化方法相关的属性都应该测试和确认。

在设计多重继承初始化相关的测试中，第一个相关的是默认初始化导致的对象属性遗忘或者误用错误。程序 6-17 首先定义了三个类 A、B、C，同时类 D1、D2、D3 根据不同的方式继承了类 A、B、C。

程序 6-17　多重继承默认初始化方法的测试

```
class A:
    def __init__(self,x):
        self.x=x

class B:
    def __init__(self,y):
        self.y=y

class C(B):
```

```
    def __init__(self,z):
        self.z=z

class D1(A,B,C):    pass
class D2(B,A,C):    pass
class D3(C,B,A):    pass

d1=D1(1)
d2=D2(2)
d3=D3(3)
print d1.x
print d2.y
print d3.z

#print d1.y
#print d1.z
#print d2.x
#print d2.z
#print d3.x
#print d3.y
```

程序 6-17 中 D1、D2、D3 无论在定义形式上还是初始化方式上几乎都非常相似。这种相似性，可能会导致属性的误用。d1 拥有对象属性 x，而 d2 拥有对象属性 y，d3 拥有对象属性 z。由于遗忘类的对应关系，或者误认为继承所有类的对象属性，均会导致错误。程序 6-17 后面 6 个带注释的 print 语句展示了这类错误。例如，执行 print d1. y 语句，Python 将提示错误信息：AttributeError：D1 instance has no attribute 'y'。

在涉及多重继承初始化相关测试中，第二个相关的是采用 super 方法调用父类方法导致的初始化错误。由于 Python 要求必须显式调用父类的初始化函数，若采用 super 调用父类的初始化方法，在多重继承过程中，也会存在类似的问题。程序 6-18 给出了测试利用 super 多重继承时初始化循序测试的例子。类 C 和类 D 分别继承自 A 类和 B 类，但是其继承的顺序不同。C 类继承的是 A 类在前，B 类在后。而 D 继承的顺序恰好相反，B 类在前而 C 类在后。这个差异导致了 C 类调用 A 类的初始化函数，而 D 类调用 B 类的初始化函数。C 类在初始化时只能接收一个参数，例如 super(C，self). __init __(1)，若更换成带注释的一行 super(C，self). __init __(2，3)，则系统出现提示：TypeError：__init __() takes exactly 2 arguments (3 given)，而在 D 类中只能接收两个参数。

程序 6-18　super 调用父类初始化顺序测试

```
class A(object):
    def __init__(self,a):
        print a

class B(object):
    def __init__(self,a,b):
```

```
        print a+b

class C(A,B):
    def __init__(self):
        super(C,self).__init__(1)
        #super(C,self).__init__(2,3)

class D(B,A):
    def __init__(self):
        super(D,self).__init__(2,3)
        #super(D,self).__init__(1)
c=C()
d=D()
```

在上面的例子中，由于参数个数的不一致，在运行到相关的语句时，系统会直接给出提示信息，错误比较容易发现。若是将其更改为带 * 的不定参数，那么在编译时就不容易发现其中存在的错误。程序 6-19 给出了一个具体的例子。

程序 6-19　隐蔽的父类初始化调用错误测试

```
#
class A(object):
    def __init__(self, * values):
        self.sum=0
        for i in range(0,len(values)):
            self.sum=self.sum+values[i]
        self.sum=self.sum/1000

class B(object):
    def __init__(self, * values):
        self.sum=0
        for i in range(0,len(values)):
            self.sum=self.sum+values[i]
        self.sum=(self.sum+1)/1000

class C(A,B):
    def __init__(self, * values):
        super(C,self).__init__(* values)

class D(B,A):
    def __init__(self, * values):
        super(D,self).__init__(* values)
c=C(2,3,4)
d=D(2,3,4)
c1=C(400,599)
```

```
d1=D(400,599)
print c.sum
print d.sum
print c1.sum
print d1.sum
```

这个程序的运行结果为：0,0,0,1。

为了集中展示该错误的隐蔽性，两个父类 A 和 B 的形式和运行结果都非常类似，但是其中存在一点儿非常小的差异。类 C、类 D 依据不同的次序继承了类 A 和类 B。只有当累加的结果大于 999 时才出现差异。如果简单地选择几个比较小的数进行累加，可能都无法发现该错误。

在涉及多重继承初始化相关测试中，第三个相关的是混用 super 和父类初始化，而导致的初始化错误。由于两种不同的调用父类初始化方法的策略完全不同，导致错误。

程序 6-20 给出了一个两种混合调用父类初始化的具体例子，其中左边一列是源代码，右边一列是运行结果。

程序 6-20　两种不同调用父类初始化方法导致的错误

```
#source code
class A(object):
    def __init__(self):
        print "enter A"
        print "leave A"

class B(object):
    def __init__(self):
        print "enter B"
        print "leave B"

class C(A):
    def __init__(self):
        print "enter C"
        super(C, self).__init__()
        print "leave C"

class D(A):
    def __init__(self):
        print "enter D"
        super(D, self).__init__()
        print "leave D"
class E(B, C):
    def __init__(self):
        print "enter E"
```

```
        B.__init__(self)
        C.__init__(self)
        print "leave E"

class F(E, D):
    def __init__(self):
        print "enter F"
        E.__init__(self)
        D.__init__(self)
        print "leave F"

f=F()

#result

enter F
enter E
enter B
leave B
enter C
enter D
enter A
leave A
leave D
leave C
leave E
enter D
enter A
leave A
leave D
leave F
```

程序 6-20 对应的类继承如图 6-1 所示，其中，类 A 和类 B 继承于 object，而类 C、类 D 继承于类 A，类 E 继承于类 B、类 C，最后类 F 继承于类 E 和类 D。类 C、类 D 采用的是 super 方式调用父类初始化方法，而类 E 和类 F 采用类名的方式调用父类的初始化方法。从右边的输出结果可以发现，类 D 和类 A 的初始化方法都被执行了两次。显然这不是实际所需要的结果。

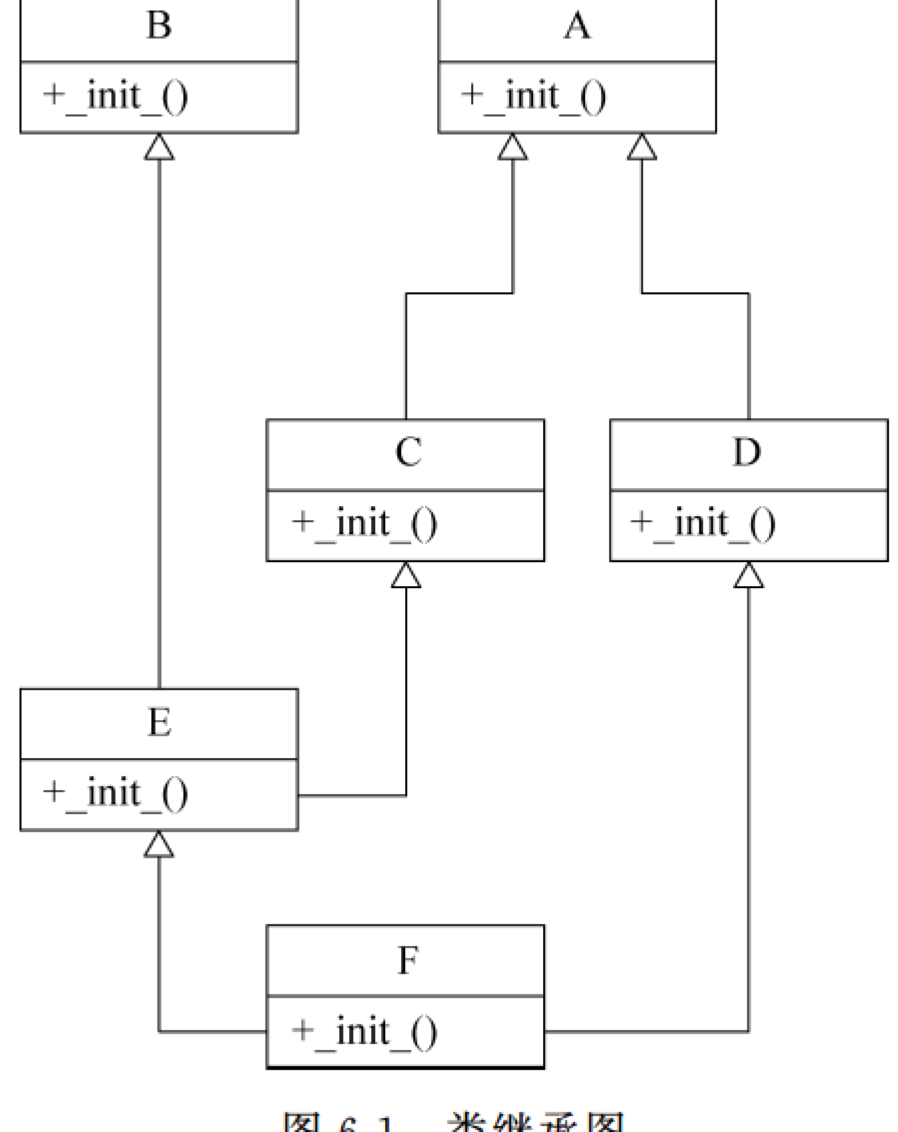


图 6-1　类继承图

6.4.3 多重继承方法解释顺序的测试

Python 中的类分为经典类和新式类。经典类是默认没有派生自某个基类的，而新式类是默认派生自 object 这个基类的。在多重继承中，如果有多个父类定义同样名字的方法，一个子类调用了父类的该方法。有两个不同的策略，新式类为 MRO（Method Resolution Order，方法解析顺序），经典的类根据深度优先的策略寻找父类对应的方法。

在新式类中，根据 C3 算法寻找父类对应的方法。C3 算法：MRO 是一个有序列表 L，在类被创建时就计算出来。L 的计算规则如下：

L(Child(Base1,Base2))=[Child+merge(L(Base1),L(Base2),Base1Base2)]

在这个算法中，其核心是 merge 操作，如果一个序列的第一个元素，是其他序列中的第一个元素，或不在其他序列中出现，则从所有执行 merge 操作的序列中删除这个元素，合并到当前的 MRO 中。merge 操作后的序列，继续执行 merge 操作，直到 merge 操作的序列为空。如果 merge 操作的序列无法为空，则说明不合法。

关于 MRO 计算过程，限于篇幅不再展开讨论。但是 MRO 直接可以通过类的 mro() 方法获取。例如查看类 A 的 MRO 列表，可以通过命令：

```
A.mro()
```

由于多重继承存在的多重解释顺序，导致调用父类的方法错误，其故障模式和测试要点如下。

故障模式：没有留意多重继承中父类方法寻找策略而导致了方法继承错误。

测试要点：无论父类的方法是否已经测试，在多重继承中如果存在多个父类重名方法的调用，所有重名方法都应该单独测试和确认。

程序 6-21 给出了一个例子。类 A 和类 C 都具有一个同样的方法 func1，并且参数的形式也完全相同。类 B 和类 C 均继承了类 A，而类 D 继承了类 B 和类 C。

程序 6-21 基于方法解析顺序 MRO 的测试

```
#程序段 1
class A():
    def func1(self):
        print "A"
class B(A):
    def func 2(self):
        pass
class C(A):
    def func 1(self):
        print "C"
class D(B, C):
    pass
d=D()
d. func1()
#程序段 2
class A(object):
```

```
    def func1(self):
        print "A"
class B(A):
    def func 2(self):
        pass
class C(A):
    def func 1(self):
        print "C"
class D(B, C):
    pass
d=D()
d.func1()
```

在程序段 1 的类的继承中，用户期望调用类 C 的 func1()函数，而实际上这段程序运行的结果显示字母 A，表明其调用了类 A 的 func1()函数，而非用户所期望的类 C。在这段程序中，类 A 没有继承任何的类，所以其属于经典类。经典类的查找顺序采用从左到右深度优先的规则，在访问 d.func1()的时候，D 这个类是没有方法 func1()，Python 会继续往其父类中寻找，但是 B 类中并不包含 func1()函数。其依据深度优先，继续在 B 的父类中寻找 func1()函数，找到了 func1()函数。直接调用类 A 的 func1()，从而导致 C 重写 A 的 func1()函数被绕过。

若将类 A 改为新型类，那么其输出的字母 C，意味着其调用的是类 C 的方法。

6.4.4 基于对象销毁的测试

在 Python 中，一切都是对象，通过引用访问相关的对象。Python 使用引用计数来追踪内存中的对象。内部记录着所有使用中的对象各有多少引用。当对象被创建时，实际上创建了一个引用计数，当对象的引用计数变为 0 时，它被垃圾回收。

程序 6-22 给出了一个包含__del__()的示例。

程序 6-22 对象销毁__del__示例

```
#coding=utf-8
#class_desctroy.py
import sys
class Employee:
    def __init__(self,name,age):
        self.name=name
        self.__age=age

    def __del__(self):
        print "destroyed"

p1=Employee('clz', 42)
```

```
print sys.getrefcount(p1)
p2=p1
print sys.getrefcount(p2)
p3=p1
print sys.getrefcount(p3)
del p1
print sys.getrefcount(p3)
del p2
print sys.getrefcount(p3)
del p3
```

程序的运行结果为：

```
2
3
4
3
2
destroyed
```

运行这段程序，仅出现一个提示信息“destroyed”。在这个示例中，Employee 类定义了__del__()函数，该函数仅给出了一个提示信息“destroyed”。在 Employee 类定义后，一个类实例化函数后面有两个赋值函数，后面附带有三个 del 函数。p1，p2，p3 三个分别指向同一个引用，这个信息可以由系统方法 sys. getrefcount()函数进行查询。由于该函数自身的调用也需要引用，所有显示的引用比看上去多 1，例如，在执行 p1＝Employee('clz', 42)以后的第一个 print sys. getrefcount(p1)语句的结果为 2。在执行前两个 del 函数时，仍然存在对对象的引用。执行 del p3 语句时，所有执行对象的引用都不再存在，Python 调用__del__()函数。

当 Python 的某个对象的引用计数降为 0 时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾，由 Python 自动回收引用计数为 0 的对象。

类对象之间的引用会存在一种环状的结构。一种最简单的方式是一个对象存在一个对象属性，该对象属性引用自己。程序 6-23 给出了一个自我引用的具体例子。

程序 6-23　自我引用而形成的引用环

```
#coding=utf-8
import gc
import sys

class A:
    pass

def testselfref():
    a=A()
    a.next=a
```

```
del a
    gc.collect()
    print gc.garbage

if __name__=="__main__":
    testselfref()
```

这个程序的运行结果为：

```
[]
```

在这个程序中，在定义了类 A 并生成了实例 a 以后，添加了对象属性 a. nextb 并指向自身。在一个存在环状结构的对象引用中，如果没有外部的对象指向该环，而仅存在于环上对象的相互引用。在执行 del a 命令以后，强制执行垃圾回收命令 gc. collect()，在此以后将系统的垃圾显示出来。从运行结果看，系统输出为空列表[]，表明 Python 已经自动识别了该应用环并删除，所有系统并不存在因此而形成的垃圾。

在存在引用环的情况下，如果用户自定义销毁了__del__()方法，Python 将无法执行回收动作引发内存泄漏。在这种情况下的错误模式和测试要点如下。

错误模式：程序中存在__del__()函数，并存在引用环而导致对象无法回收。

测试要点：寻找引用环，并设法删除环中的一条引用。

若一个类存在__del__方法，该方法中很可能使用类中的相关引用，但如果在之前的 del a 时将 a 给回收掉，此时将造成异常。所以 Python 没办法进行自动回收，造成了 uncollectable，也就产生了内存泄漏。所以__del__方法要慎用，如果用的话一定要保证没有循环引用。

程序 6-24 给出了用户自定义__del__()方法，并引用自身导致内存泄漏的例子。

程序 6-24　对象属性引用自身导致的内存泄漏

```
#coding=utf-8
#classleak.py
import gc
import sys

class Node(object):
    def __init__(self):
        print 'Object (id % d) created.' %  id(self)
    def __del__(self):
        print 'Object (id % d) destroyed.' %  id(self)

def fool():
    a=Node()
    a.next=a
    del a
    gc.collect()
```

```
    print gc.garbage

if __name__=="__main__":
    foo1()
```

这个程序的执行结果为：

```
Object (id 46217936) created.
[<__main__.A instance at 0x02C13AD0>]
```

这个运行结果表示程序在 del a 语句以后，并没有调用__del__()方法。在这个例子中，在进行实例化以后产生对象 a，动态给对象 a 添加对象属性。而该对象属性是一个指向自身的引用。

所以在执行 del a 语句以后，由于用户自定义了__del__()方法，导致对象无法回收。

若存在两个对象，并引用对方，也会构成一个引用环。引用环的存在会给上面的垃圾回收机制带来很大的困难。这些引用环可能构成无法使用但引用计数不为 0 的一些对象。程序 6-25 给出了一个孤立引用环的例子。

程序 6-25　孤立的引用环导致的内存泄漏

```
#coding=utf-8
#classleak.py
class Node(object):
    def __init__(self):
        print 'Object (id %d) created.' % id(self)
    def __del__(self):
        print 'Object (id %d) destroyed.' % id(self)

def foo():
    a=Node()
    b=Node()
    a.next=b
    b.next=a
    del a
    del b
if __name__=="__main__":
    foo()
```

这个程序执行的结果为：

```
Object (id 44445488) created.
Object (id 44445648) created.
```

和前面的例子类似，这个程序也没有调用对象__del__()方法，也意味着其占用的内存并未被释放。

程序分别创建 Node 的两个对象，其中 a 对象属性 a.next 指向了 b，而对象 b 的对象属性 b.next 指向了 a。构成了一个典型的孤立环。必须消除这个引用环，才能使得引用

为 0。将程序 6-25 进行修改，删除两个对象的相互引用的任意一套引用，如程序 6-26 所示。

程序 6-26 删除引用环以后对象内存自动回收

```
#coding=utf-8
class Node(object):
    def __init__(self):
        self.x=5
        print 'Object (id % d) created.' %  id(self)
    def __del__(self):
        print 'Object (id % d) destroyed.' %  id(self)

def foo1():
    c=Node()
    c.next=c
    del c

def foo():
    a=Node()
    b=Node()
    a.next=b
    b.next=a
    del a.next
    print b.x
    print a.x
if __name__=="__main__":
    foo()
```

这个程序的运行结果如下。

```
Object (id 44445488) created.
Object (id 44445648) created.
5
5
Object (id 44445648) destroyed.
Object (id 44445488) destroyed.
```

在这个例子中，构成结束之前先删除其中的任意一条引用，在程序结束前，可以看到对象都还存在，通过打印两个对象属性 x，显示正常。显然从运行结果可以看出，在运行结束以后，两个对象都已经被释放。

6.5 基于装饰器的测试

在实际的业务实现中，存在大量和核心业务无关的功能，如引入日志、增加计时逻辑来检测性能、给函数增加事务的能力。这些功能的存在使得核心业务的表达变得非常难

理解。装饰器可以使得将与业务无关的代码从系统中切割出来进行单独编程。

在 Python 中,一切都是类,都是 object 的子类。函数也是一个类,是 object 的子类。通过下面的例子说明。

```
#coding=utf-8
#funisclass.py
def foo(func):
    func()

print issubclass(foo.__class__,object)

def bar():
    print "I am in bar"

foo(bar)
```

装饰器从本质上就是函数,是用来包装函数的函数,用来修饰原函数,将其重新赋值给原来的标识符,并永久地丧失原函数的引用。

在介绍装饰器之前,先介绍一个非常重要的特性:“闭包”。在面向对象的范式中,对象是附有行为的数据,而闭包是附有数据的行为。Python 中的闭包从表现形式上解释为:如果在一个内部函数里,对在外部作用域(但不是在全局作用域)的变量进行引用,那么内部函数就被认为是闭包。程序 6-27 给出了闭包和普通函数之间的比较。

程序 6-27　闭包和普通函数的比较

```
#coding=utf-8

def func(x):
    y=x
    def __func():
        a=1
        return 1
    return __func

def cfunc(x):
    b=1
    def __cfunc():
        y=b+x+1
        return x
    return __cfunc
```

在这个例子中,尽管__func(x)和__cfunc(x)都是内嵌在外面的函数内部,__func(x)是一个内嵌的普通函数,而__cfunc(x)是一个闭包。因为__cfunc(x)直接使用了外部变量 x(相对于函数__cfunc 而言,而不是相对 cfunc 而言),而__func(x) 未使用任何的外部变量。

对于闭包而言，在闭包函数的生存周期内，闭包变量从效果上类似全局变量。程序6-28给出了闭包函数对于闭包变量的影响。此时故障模式和测试要点如下。

故障模式：遗忘变量在闭包函数生存周期内的全局性，遗忘上次执行对于本次闭包函数执行的影响。

测试要点：如果在函数中存在更改过的闭包变量至少要测试两次，以检查多次闭包函数调用对闭包变量值的影响。

程序 6-28 闭包函数对于闭包变量的影响

```
#coding=utf-8
#closure_demo7.py
def fun():
    count=[3]
    def counter():
        count[0]-=1
        result=round(1/count[0])
        print "1/%s is % s"%(count[0],result)
    return counter

myfunc=fun()
myfunc()
myfunc()
myfunc()
```

其运行结果如下。

```
1/2 is 0.0
1/1 is 1.0
File "e:\2014\bookpub\software_testing\python\closure demo7.py", line 12, in<module>    myfunc()
File "e:\2014\bookpub\software_testing\python\closure demo7.py", line 5, in counter    result=round
(1/count[0])
ZeroDivisionError: integer division or modulo by zero
```

三个语句的调用形式是完全相同的，前两条语句正常运行，第三条语句出现除零错误。该函数是用于计算倒数的一个函数，需要计算的初始值存放在count列表中，每次调用以后分母的值都将减去1。但是由于闭包变量的不可见性，同时在myfunc函数的接口上也没有任何相关的信息而导致错误。

上述例子的错误比较明显，很容易就发现。但是在有些闭包函数中，对于全局变量的引用和改变比较隐蔽，发现其错误比较困难。程序6-29给出了一个例子，这个例子希望定义一系列函数，每一个函数都比前一个函数的值大1，即输出结果为[5,6,7,8]。

程序 6-29 循环变量是全局变量引起的错误

```
#coding=utf-8
#closure_demo8.py
fs=[]
```

```
for i in range(4):
    def f(n):
        return i+n
    fs.append(f)

print[f(5) for f in fs]
```

实际上,这个程序运行的结果为:

```
[8,8,8,8]
```

在 Python 中,循环变量 i 是一个全局变量,所有的闭包函数都指向该变量。当循环结束时,变量 i 的值为 3,保存在 fs 列表中的所有函数的内容都是:3+n。显然当执行 f(5)时,其结果都是 8。

显然如果对于这个程序中的 f 闭包,仅调用一次是无法发现该错误的,参见程序 6-30。

程序 6-30　调用一次闭包函数无法发现错误

```
#coding=utf-8
#closure_demo8.py
fs=[]

for i in range(1):
    def f(n):
        return i+n
    fs.append(f)

print[f(5) for f in fs]
```

该程序运行的结果为:

```
[5]
```

在仅调用一次的情况下,两者刚好相等,测试无法反映出全局变量修改的结果的影响。

若要消除闭包对于全局数据的影响,可以将该闭包修改成一个函数。程序 6-31 给出了一个具体的例子。

程序 6-31　修改成参数使得变量局部化

```
#coding=utf-8
#closure_demo8.py
fs1=[]
for i in range(4):
    def f(n,i=i):
        return i+n
```

```
        fs1.append(f)

print[f(5) for f in fs1]
```

该程序运行的结果为：

```
[5,6,7,8]
```

闭包的重要用途是装饰器，装饰器的功能是在函数执行的前后可以附加额外的功能。程序 6-32 给了一个装饰器的例子，该装饰器计算一个函数具体的执行时间。

程序 6-32　实现装饰器的例子

```
#-*-coding:UTF-8-*-
#closuredemo8timeit1.py
import time

def timeit(func):
    def wrapper():
        start=time.clock()
        func()
        end=time.clock()
        print 'used:', end-start
    return wrapper

def foo():
    print 'in foo()'
foo=timeit(foo)

foo()

#-*-coding:UTF-8-*-
#closuredemo8timeit1.py
import time

def timeit(func):
    def wrapper():
        start=time.clock()
        func()
        end=time.clock()
        print 'used:', end-start
    return wrapper

@ timeit
def foo():
    print 'in foo()'
foo()
```

这个程序运行的结果为：

```
in foo()
used: 4.76667143334e-05
```

其中，foo 函数是被装饰的函数，timeit 是装饰函数。timeit 在函数 foo 执行的前后分别进行计时，并输出时间差，显然 timeit 内部就是一个典型的闭包，并返回一个函数，该函数的接口参数和 foo 完全相同，foo＝timeit(foo)实现了用 timeit 函数去替代原来的 foo 函数。其后执行的所有 foo 函数，实际上都是执行的 timeit 函数。在左边部分，为了简化书写，Python 提供了@语法糖，在实际效果上左边和右边是完全相同的。在这种情况下，主要的故障模式和测试要点如下。

故障模式：在设计测试时仅测试被装饰函数或者仅测试装饰函数。

测试要点：必须将装饰函数和被装饰函数综合起来考核测试需求。

在上述计时程序中，装饰器和被装饰函数几乎是独立的，有时装饰器和被装饰函数有较强的相互影响，例如装饰器作为前面运算的缓存功能。

程序 6-33 给出了一个利用缓存装饰器的斐波那契数列，它利用一个具有字典功能的字典 memo 存储已经计算过的 fib(n)函数值，其中 n 为关键字，而 fib(n)为计算值。在每一次调用之前，先检查和 n 对应的 fib(n)值是否已经存在，如果不存在执行传统 fib 递归调用，如果曾经计算过，那么直接调用 memo 存储的值。

程序 6-33　具有缓存装饰器的斐波那契数列

```
def memoize(f):
    memo={}
    def helper(x):
        if x not in memo:
            memo[x]=f(x)
        return memo[x]
    return helper

@memoize
def fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n-1)+fib(n-2)

#fib=memoize(fib)

print(fib(40))
```

程序 6-34 给出了一个利用装饰器实现缓存的斐波那契数列的例子。和前一个例子

的主要差别是：类 Memoize 是一个装饰器类而不是一个函数。当将一个类作为装饰器时，需要重新定义类的__call__()方法。该方法就是直接使用类名进行调用时执行的方法。

程序 6-34　具有缓存装饰器类的斐波那契数列

```
#decrator_demo5_fib.py
class Memoize:
    def __init__(self, fn):
        self.fn= fn
        self.memo= {}
    def __call__(self, * args):
        if args not in self.memo:
            self.memo[args]= self.fn(* args)
        return self.memo[args]


@Memorize
def fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n- 1)+ fib(n- 2)


print(fib(7))
```

在这个程序中，fib()函数是经典的递归调用函数，但是递归调用函数执行的效率比较低，存在多次重复调用的情况。图 6-2(a)的流程给出了一个 fib(5)调用的例子，其中 f(4)被重复调用了一次，fib(3)被重复调用了两次，fib(2)被重复调用了三次，fib(1)被重复调用了五次。具有缓存功能的流程图如图 6-2(b)所示，在具有装饰器缓存的流程中，利用装饰器将每一次调用的结果缓存在 memo 中，每一个函数最多被调用一次。由于装饰器的加入，改变了原来程序执行的流程。

对于黑盒测试而言，无论是否具有装饰器，其测试方法都是没有差别的。而对于白盒测试，在具有装饰器的函数中，可以将装饰函数中的被装饰函数展开，参考前面控制流设计测试。

程序 6-35 是将装饰器展开以后对应的程序。显然在这个程序中，假设采用分支覆盖准则，那么可以得出以下 4 个基本的用例。

fib2(0)：覆盖 n==0 所在的分支。

fib2(1)：覆盖 n==1 所在的分支。

fib2(2)：覆盖 memo2[n]=fib2(n−1)+fib2(n−2)所在的分支。

fib2(3)：覆盖缓存的情况。

程序 6-35　将装饰器展开以后的对应程序

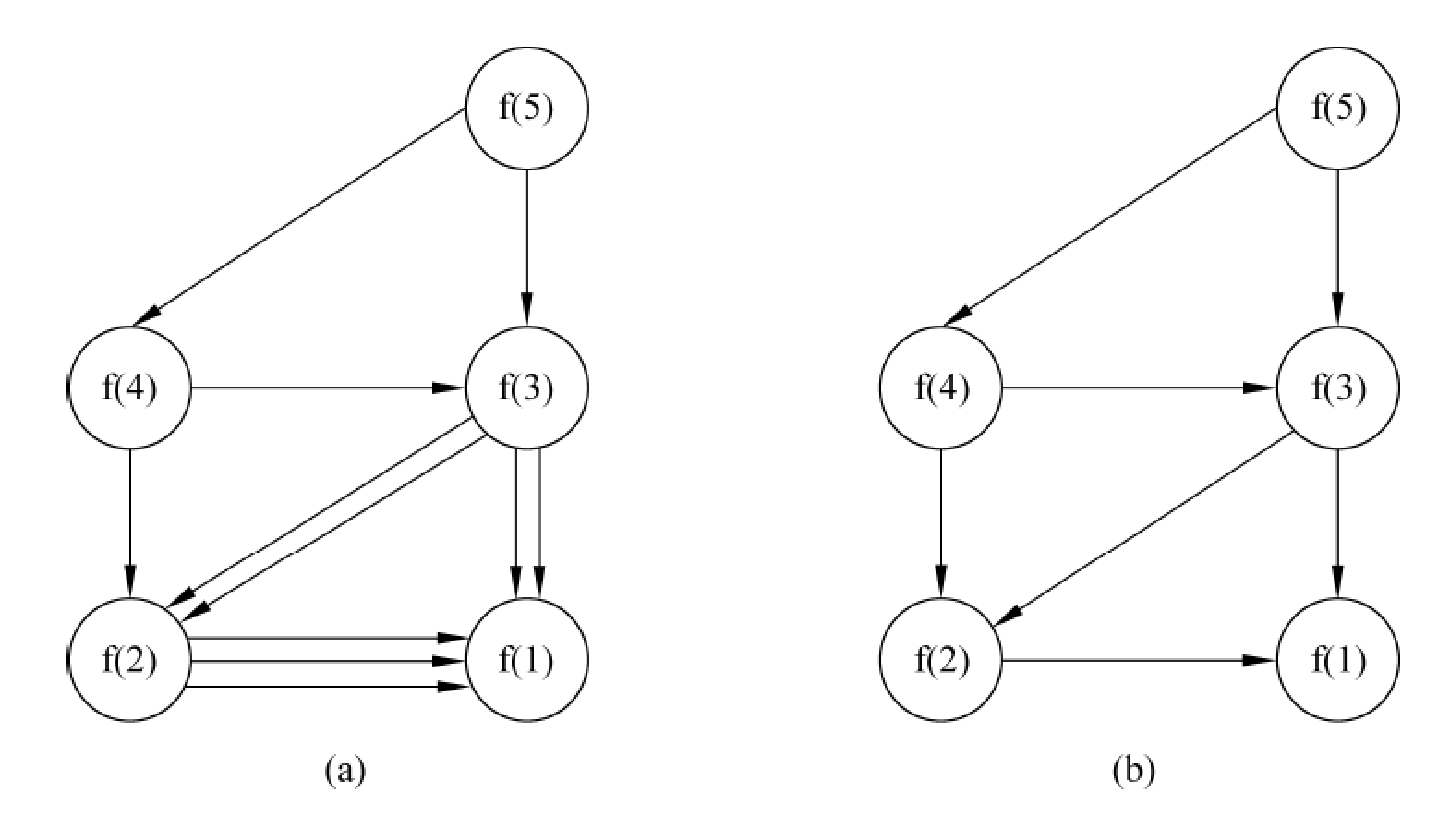


图 6-2　具有缓存和不具缓存功能执行流程的改变

```
memo2={}
def fib2(n):
    if n not in memo2:
        if n==0:
            return 0
        elif n==1:
            return 1
        else:
            memo2[n]=fib2(n-1)+fib2(n-2)
            return memo2[n]
    else:
        return memo2[n]

print(fib2(40))
```

6.6　基于多态的测试

多态是指在面向对象中能够根据使用类的上下文来重新定义或改变类的性质和行为。在实际运行时,父类就可以根据当前赋值给它的子类的特性以不同的方式运作。

多态是面向对象的基本特征,一般实现多态,需要具备以下几个条件。

(1) 有继承。

(2) 有重写。

(3) 父类引用指向子类对象。

在 Python 中,由于一切都是对象,并且是动态类型,凡是具有同样方法的都可以实现类似多态的功能。多态提供很好的程序可扩展性,当新增加一个同类对象时,凡是以其父类引用为参数的函数或者方法均可以不必修改。

程序 6-36 给出了多态的具体例子。

程序 6-36　一个喂养动物的多态程序的例子

```
#Python 实现多态实例

class Animal(object):
    def __init__(self):
        pass
    def Eat(self):
        pass

class Chicken(Animal):
    def __init__(self,name):
        self.name= name
        super(Chicken, self).__init__()
    def Eat(self):
        print 'the chicken %s has been eat'%self.name

class Dog(Animal):
    def __init__(self,name):
        self.name= name
        super(Dog, self).__init__()
    def Eat(self):
        print 'the dog %s has been eat'% self.name

class Person(object):
    def __init__(self,name):
        self.name= name
        self.animals= []

    def addpet(self,pet):
        self.animals.append(pet)

    def Feed(self):
        for pet in self.animals:
            pet.Eat()

def test():
    Kobe= Person('Kobe')
    Kobe.addpet( Chicken("ChickenA"))
    Kobe.addpet(Dog("DogA"))
    Kobe.addpet(Dog("DogB"))

    Kobe.Feed()
```

```
if __name__=="__main__":
    test()
```

这个程序的运行输出结果为：

```
the chicken ChickenA has been eat
the dog DogA has been eat
the dog DogB has been eat
```

在这个例子中，Chicken 和 Dog 都是 Animal 的子类，并且它们重写了父类的 Eat()函数。Person 类中有两个方法和 Animal 类相关，其中，addpet()表示 Person 家增加新的宠物，而 Feed 表示其喂养所有的宠物。此时并不知道要喂养的是什么宠物，显然 Chicken 和 Dog 的喂养方法是不一样的，但是没有必要去关注这些差别。

如果 Person 喂养了一种新类型的宠物，例如 Cat，由于 Person 类中所调用的是其父类 Animal，Person 无须做任何的修改，可以增加新的宠物种类，大大增加了程序的可扩展性。

程序 6-37 在前一个程序的基础上，增加了新喂养的动物 Cat，同样 Cat 也是继承自类 Animal，但是其也有其独特的 Eat()方法。在这种情况下，无须修改 Person 类和 Animal 相关的任何方法。

程序 6-37　增加了一种新喂养动物的多态程序

```
#Python实现多态实例

class Animal(object):
    def __init__(self):
        pass
    def Eat(self):
        pass

class Chicken(Animal):
    def __init__(self,name):
        self.name=name
        super(Chicken, self).__init__()
    def Eat(self):
        print 'the chicken %s has been eat'%self.name

class Dog(Animal):
    def __init__(self,name):
        self.name=name
        super(Dog, self).__init__()
    def Eat(self):
        print 'the dog %s has been eat'%self.name

class Cat(Animal):
```

```
    def __init__(self,name):
        self.name=name
        super(Cat, self).__init__()
    def Eat(self):
        print 'the cat %s has been eat'%self.name

class Person(object):
    def __init__(self,name):
        self.name=name
        self.animals=[]

    def addpet(self,pet):
        self.animals.append(pet)

    def Feed(self):
        for pet in self.animals:
            pet.Eat()

def test():
    Kobe=Person('Kobe')
    Kobe.addpet( Chicken("ChickenA"))
    Kobe.addpet(Dog("DogA"))
    Kobe.addpet(Dog("DogB"))
    Kobe.addpet(Cat("CatA"))

    Kobe.Feed()

if __name__=="__main__":
    test()
```

这个程序的运行输出结果为：

```
the chicken ChickenA has been eat
the dog DogA has been eat
the dog DogB has been eat
the cat CatA has been eat
```

在前面的例子中，类 Person 中的 Feed 方法实际上根据 animals 中对象类型的不同，分别调用了不同的方法，实际上等效于一个多重判断语句。程序 6-38 给出了一个和多态等价的利用 if 语句实现的 Feed()方法，显然这是一个由多个 if 语句构成的代码段。利用 if 语句判断当前的变量是属于哪一个类的实例，然后调用不同的方法。在程序中，通过为一个变量赋值，仅为了在字面上说明这个意思，从语法上并没有这个必要。

程序 6-38　和多态等价的 if 判断语句

```
#Python实现多态实例
```

```
class Animal(object):
    def __init__(self):
        pass
    def Eat(self):
        pass

class Chicken(Animal):
    def __init__(self,name):
        self.name=name
        super(Chicken, self).__init__()
    def Eat(self):
        print 'the chicken %s has been eat'%self.name

class Dog(Animal):
    def __init__(self,name):
        self.name=name
        super(Dog, self).__init__()
    def Eat(self):
        print 'the dog %s has been eat'%self.name

class Cat(Animal):
    def __init__(self,name):
        self.name=name
        super(Cat, self).__init__()
    def Eat(self):
        print 'the cat %s has been eat'%self.name

class Person(object):
    def __init__(self,name):
        self.name=name
        self.animals=[]

    def addpet(self,pet):
        self.animals.append(pet)

    def Feed(self):
        for pet in self.animals:
            if  isinstance(pet, Chicken):
                chicken=pet
                chicken.Eat() #调用 Chicken的 Eat方法

            if  isinstance(pet, Dog):
                dog=pet
```

```
                dog.Eat() #调用 Dog 的 Eat 方法

            if  isinstance(pet, Cat):
                cat=pet
                cat.Eat()#调用 Cat 的 Eat 方法

def test():
    Kobe=Person('Kobe')
    Kobe.addpet( Chicken("ChickenA"))
    Kobe.addpet(Dog("DogA"))
    Kobe.addpet(Dog("DogB"))
    Kobe.addpet(Cat("CatA"))

    Kobe.Feed()

if __name__=="__main__":
    test()
```

这个程序执行的结果为：

```
the chicken ChickenA has been eat
the dog DogA has been eat
the dog DogB has been eat
the cat CatA has been eat
```

基于控制流的测试方法的章节中已经非常详细地讨论过分支和路径测试的方法，在分支方法中，每一个 if 判断都必须至少被一个测试用例所覆盖。将这个思想移植到多态方法中，可以得出一个重要的覆盖准则。

在具有一个多态的类方法中，如果增加了一个多态处理的子类，那么在该方法的测试中必须增加一个测试用例以覆盖该子类。或者说多态处理的类中的每一个子类至少被一个测试用例所覆盖。

第 7 章　基于 UML 的软件测试

UML 是一个支持模型化和软件系统开发的图形化语言，为软件开发的不同阶段提供模型化和可视化支持。UML 提供了多种类型的模型描述图，最常用的 UML 图包括：用例图、类图、序列图、状态图、活动图、组件图和部署图等。UML 在作为系统建模工具的同时，也逐步成为软件测试设计的依据。本章讨论面向对象和基于 UML 模型的测试的相关问题。

7.1　UML 概念和建模

在工程实践中，人们看到的问题需求最直接的表现形式是过程，尽管在过程中间涉及很多对象。将这些对象抽象出来并非一件容易的事情，用户和开发者常常不能精确地理解和描述将要开发的软件系统。于是各种面向对象分析 OOA、面向对象设计 OOD、面向对象编程 OOP 应运而生。UML(Unified Modeling Language，统一建模语言)是一种图形化的建模语言，是一种面向对象分析与设计的重要工具。在统一 BOOCH、OMT、OOSE 等方法的基本概念和表示符号的基础上，1997 年 11 月，国际对象组织 OMG (Object Management Group)发布了 UML 1.0 版。2003 年 3 月，OMG 发布了 UML 2.0 版规范。

为了能够方便地处理面向对象编程，必须先对所描述的系统建立系统模型。模型从一个系统的重要特征的表现，突出系统的重要方面，而忽略其他方面。模型能够将所要设计的结构和系统的行为联系起来，并对系统的体系结构进行可视化和控制。UML 的核心要素由三部分构成：模型元素、通用机制、视图，如图 7-1 所示。UML 是一种建模语言，意味着它有自己的语法规则。UML 模型元素定义了表达构成系统的模型元素，包括构成系统的事物，以及事物之间的关系，与具体的程序开发语言和平台无关，也和软件生存周期所采用的开发过程和模型无关。通用机制包括和特定模型无关的一些机制，例如修饰、注释等。

UML 视图是表达系统的表现形式。UML 利用各种视图描述系统模型，包括系统的静态和动态特征，从不同的视角为系统架构建立模型。UML 主要的视图包括以下几种。

(1) 用户视图：以用户的观点表示系统的目标，它是所有视图的核心，该视图描述系统的需求。用户视图主要包括用例图，主要在宏观上描述系统的功能。

(2) 结构视图：表示系统的静态行为，描述系统的静态元素如包、类与对象，以及它们之间的关系。主要包括类图和对象图，类图描述系统类之间的静态结构，而对象图描述系统在某个时刻的静态结构。

(3) 行为视图：表示系统的动态行为，描述系统的组成元素如对象在系统运行时的交互关系。行为视图主要包括序列图、协作图、状态图、活动图。序列图按照时间顺序描

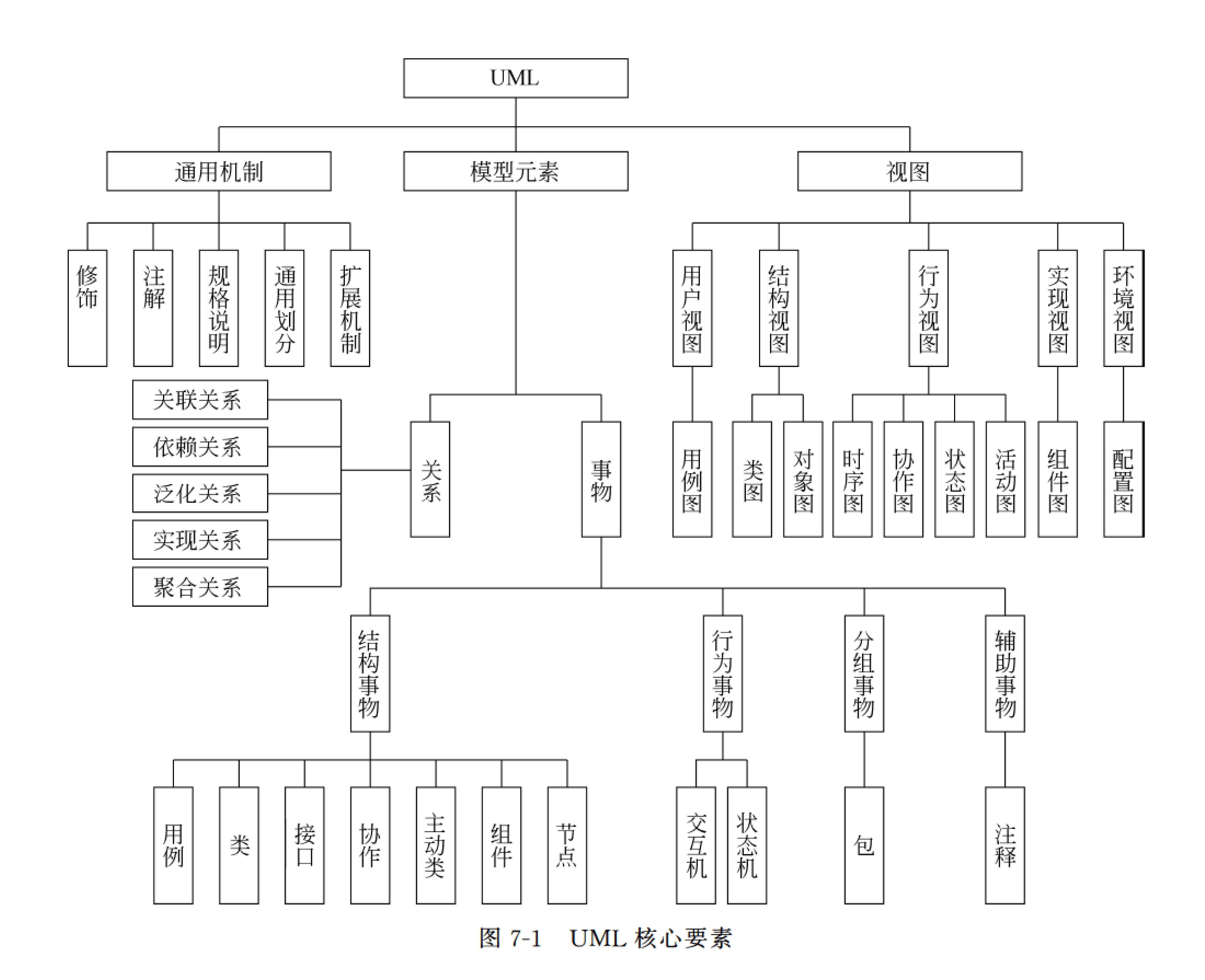


图 7-1　UML 核心要素

述系统元素间的交互；协作图根据空间和时间的顺序描述元素间的交互；状态图描述了系统元素的状态条件和响应；活动图描述系统元素的活动。

(4) 实现视图：表示系统中逻辑元素的分布，描述系统中物理文件以及它们之间的关系。包括组件图，描述系统元素的组织。

(5) 环境视图：表示系统中物理元素的分布，描述系统中硬件设备以及它们之间的关系。部署图，描述环境元素的配置，并把现实系统的元素映射到配置上。

UML 在软件生存周期中的不同环节都得到非常广泛的应用。在需求分析阶段，可使用用例图来捕获用户的需求。通过用例建模，描述对系统感兴趣的外部角色和他们对系统的功能要求。在系统分析阶段，用 UML 表达问题域中的主要概念，如对象、类以及它们之间的关系等，需要建立系统的静态模型，可用类图来描述。为了实现用例表示的功能，在不同类之间需要协作，可以用动态模型的状态图、序列图和协作图来描述。在分析阶段，只考虑问题域中的对象建模，通过静态模型和动态模型来描述系统结构和系统行为。在系统设计阶段考虑定义软件系统中的技术细节用到的类，如引入处理用户交互的接口类、处理数据的类、处理通信和并行性的类。在实现阶段，使用面向对象程序设计语言，将来自设计阶段的类转换成源程序代码，用构件图来描述代码构件的物理结构以及构件之间的关系。用配置图来描述和定义系统中软硬件的物理体系结构。在测试时，不同

阶段建立的UML模型都是测试生成和分析的依据。可使用类图、状态图进行内类单元测试,可以使用状态图和序列图、活动图、写作图进行类间的交互测试,使用组件图、协作图进行集成测试,使用用例图进行确认测试,以验证测试结果是否满足用户的需求。

7.2 基于用例的软件测试

7.2.1 用例图的概念

用例图主要用来表示系统的核心模块,从客户的视角看到系统应该具有的功能。用例图常用来表示客户的需求,即软件应该具有的功能模块以及这些模块之间的调用关系。同时,用例图通常还包含用例的执行者,用例描述了系统外部可见的行为。用例是系统的中心,它是其他视图开发的基础,用于确认和最终验证系统。用例也可以认为是对系统功能的分解描述。

用例图中,一个非常重要的角色就是执行者(Actor),这里执行者可以是人,或者在系统之外,透过系统边界与系统进行交互的其他系统,以及自动发生的事件。引入执行者,帮助系统确定其边界。使用系统的人员、改变系统的人员、使用系统的信息的人员、和系统发生交互的其他系统、自动发生的事件等都可以是执行者。

在用例图中,执行者用一个小人表示,用例表示的角色写在小人下面。

用例由执行者来激活,并提供确切的值给执行者。用例的复杂程度根据系统的需求而定,是一个具体的用户目标实现的完整描述。用例图着重于从系统外部执行者的角度来描述系统需要提供哪些功能,指明这些功能的执行者是谁;描述需求分析之后的各个阶段的工作。用例建模(Use Case Modeling)是使用用例来描述系统的功能需求的过程,用例建模促进并鼓励了用户参与,这是确保项目成功的关键因素之一。用例是用户与计算机之间为达到某个目的而进行的一次交互作用,即系统执行的一系列动作。动作执行的结果能被指定的执行者见到。用例描述了用户提出的一些可见的需求,它实现了一个具体的用户目标。

在用例图中,用例一般用一个椭圆表示,用例的名称一般放在椭圆的里面或者下面。

执行者和用例之间进行交互,相互之间的关系用一根直线来表示,称为关联关系(Association)或通信关系(Communication)。在一个ATM系统中,ATM用户执行一个取款功能,这个用例如图7-2所示。

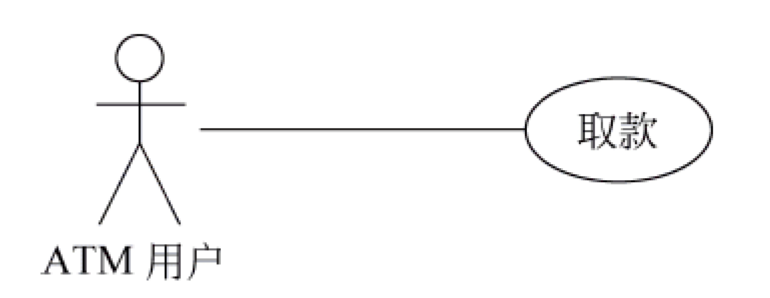


图7-2 ATM系统取款用例图

一个执行者可以参与一个或者多个用例,同样一个用例也可以和多个执行者相关联。在一个系统中,存在一些执行者,他们之间存在共享和增强。一个执行者A具有执行者B所有的性质,并在B的基础上存在若干扩展属性,这种关系称为泛化关系。例如,一个银行系统中的用户包括普通用户、银卡用户、金卡用户,不同用户的权限是不断增强的关系,凡是普通用户具有的权限,银卡用户、金卡用户均具有。执行者之间的这种关系,称为泛化(Generalization)关系(或称为“继承”关系)。执行者之间的泛化关系,用一个带有空心三角的箭头表示,其

方向由子执行者指向父执行者。图 7-3(a)表示了银卡用户和金卡用户和普通用户之间的泛化关系。而图 7-3(b)表示不同用户具有不同的功能,一个普通用户仅具有修改自己个人信息的功能,而管理员既有普通用户的功能,还具有增加用户的功能。

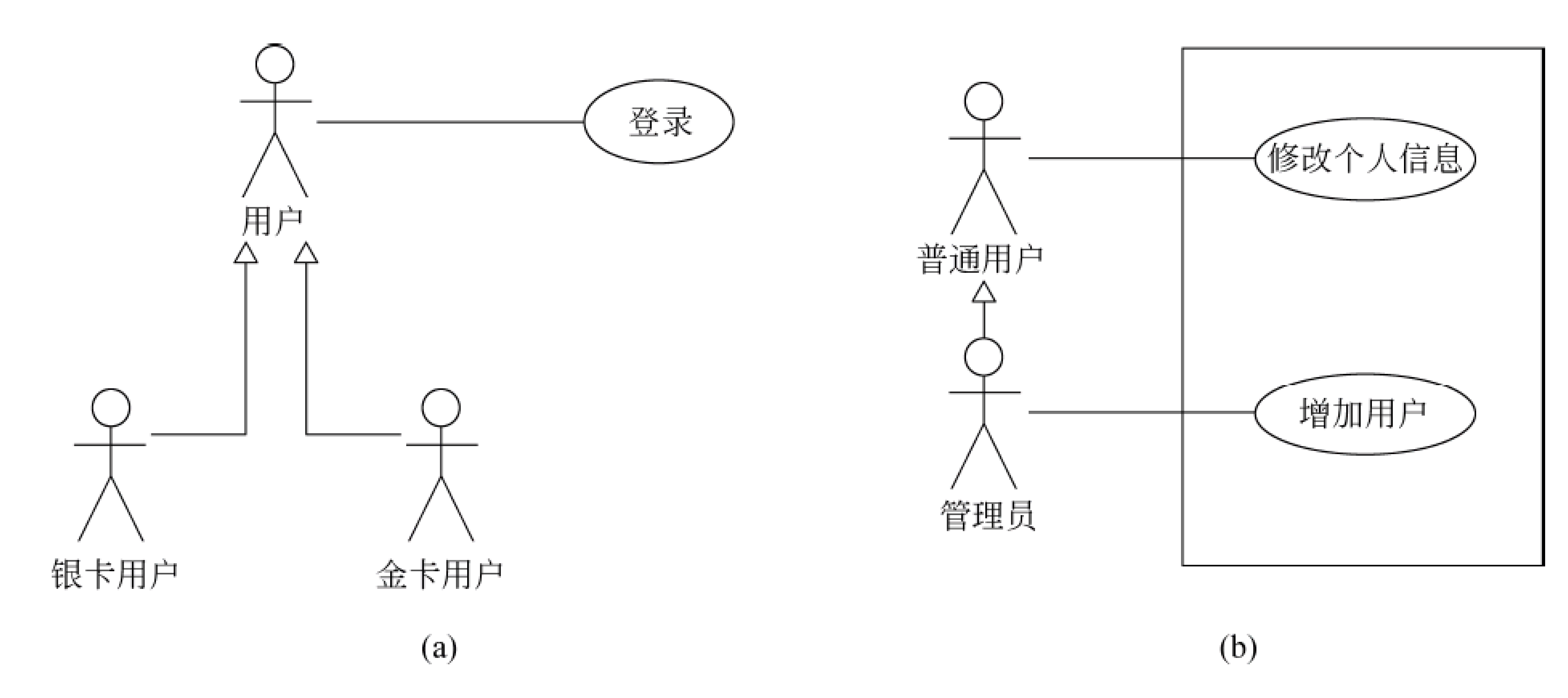


图 7-3 执行者的继承关系

用例之间的关系主要包括:包含关系,扩展关系,泛化关系。

包含关系指一个用例使用或者包含另一个用例中的行为,包含关系是通过在依赖关系上增加<<include>>构造型来表示的。由用例 A 指向用例 B,表示用例 A 中使用了用例 B 中的行为或功能。

在扩展关系中,基础用例中定义有一至多个已命名的扩展点,扩展关系是指将扩展用例的事件流在一定的条件下按照相应的扩展点插入到基础用例中。

扩展关系是通过在依赖关系上应用<<extend>>构造型(衍型)来表示的。扩展用例可以在基础用例之上添加新的行为,但是基础用例必须声明某些特定的"扩展点",并且扩展用例只能在这些扩展点上扩展新的行为。

当多个用例共同拥有一种类似的结构和行为的时候,可以将它们的共性抽象成为父用例,其他的用例作为泛化关系中的子用例。

在用例的泛化关系中,子用例是父用例的一种特殊形式,子用例继承了父用例所有的结构、行为和关系。

用例的定义包含用例所要执行的行为,包括用例的相关利益方、用例执行的前置条件、用例执行的后置条件、执行的基本路径和扩展路径、用例的扩展点。在用例的动态执行过程中,一个用例可以采用状态图、序列图、通信图或者文字来描述。用例是系统功能的描述,但并不是其实现的方法。一个类在系统中可能有多重角色,那么其可能实现各个用例的部分功能。用例是一个业务功能,而不是一个具体的操作步骤。例如,取款是一个业务功能,但是在取款过程中,用户输入密码是取款时执行者的一个具体步骤,并不适合作为一个用例。

用例建模首先需要识别执行者和用例,分析各个元素之间的关系,才能得到用例图。用例图所涉及的图标,如表 7-1 所示。

表 7-1　用例图的图标表示

名称	解　　释	图示
执行者	与系统交互的用户或者外部系统	
用例	系统的功能	
系统	描述系统的边界	
关联	用例和执行者的关联关系	
包含	一个用例包含另一个用例	
泛化	执行者之间或用例之间的泛化关系	
扩展	一个用例扩展了另一个用例	

7.2.2　用例图的覆盖准则

用例图描述了整个系统的执行者和功能,适用于基于需求的测试和更高层次的测试。基于用例图的测试应当覆盖用例图中所有的执行者和用例来保证测试的完整性。如果仅直观地利用用例图中的执行者、用例和关系来设计测试用例是不够的,还需要利用用例的详细内容。可以通过用例描述来获得用例的详细信息,其主要需要的信息包括执行者、触发动作、前置条件、后置条件、基本路径、扩展路径和扩展点。基于用例图的测试主要有以下 4 个覆盖准则,其中,前面三个准则是针对整个用例图,场景覆盖准则是针对单个用例的。

1. 执行者覆盖准则

执行者覆盖准则要求测试用例必须覆盖用例图中所有的执行者。覆盖所有执行者意味着每个执行者至少有一个用例需要进行测试。当一个执行者与多个用例相关时,可以只选择一个用例进行测试。多个执行者可能会选择同一个用例进行测试,主要有两种情况。一种情况是这个用例需要多个执行者共同完成,则只需要测试该用例一次;另一种情况是每个执行者都可以单独完成该用例,则该用例要对应每个执行者进行单独的测试,即每个执行者单独测试该用例。

以学生课程管理系统为例,学生可以在该系统上选课和查看课程成绩,教师可以提交教授课程和提交成绩,这两个角色进行操作前都需要登录。另外,有一个作业系统,学生、教师和作业系统共同进行作业管理。学生课程管理系统的用例图,如图 7-4 所示。

学生课程管理系统中有学生、教师和作业管理系统三个执行者,为了实现执行者覆盖准则,每个执行者都需要选择一个用例进行测试。学生需要在查看成绩单、选课、登录和作业管理 4 个用例中选择一个用例进行测试。教师需要在登录、作业管理、提交教授课程和提交成绩 4 个用例中选择一个用例进行测试。作业系统管理只有一个相关用例,因此只能选择作业管理用例进行测试。随机选择用例可以保证执行者的覆盖,但是可能存在冗余。从图 7-4 中可知,用例作业管理与学生、教师和作业管理系统都有关联,并且这三

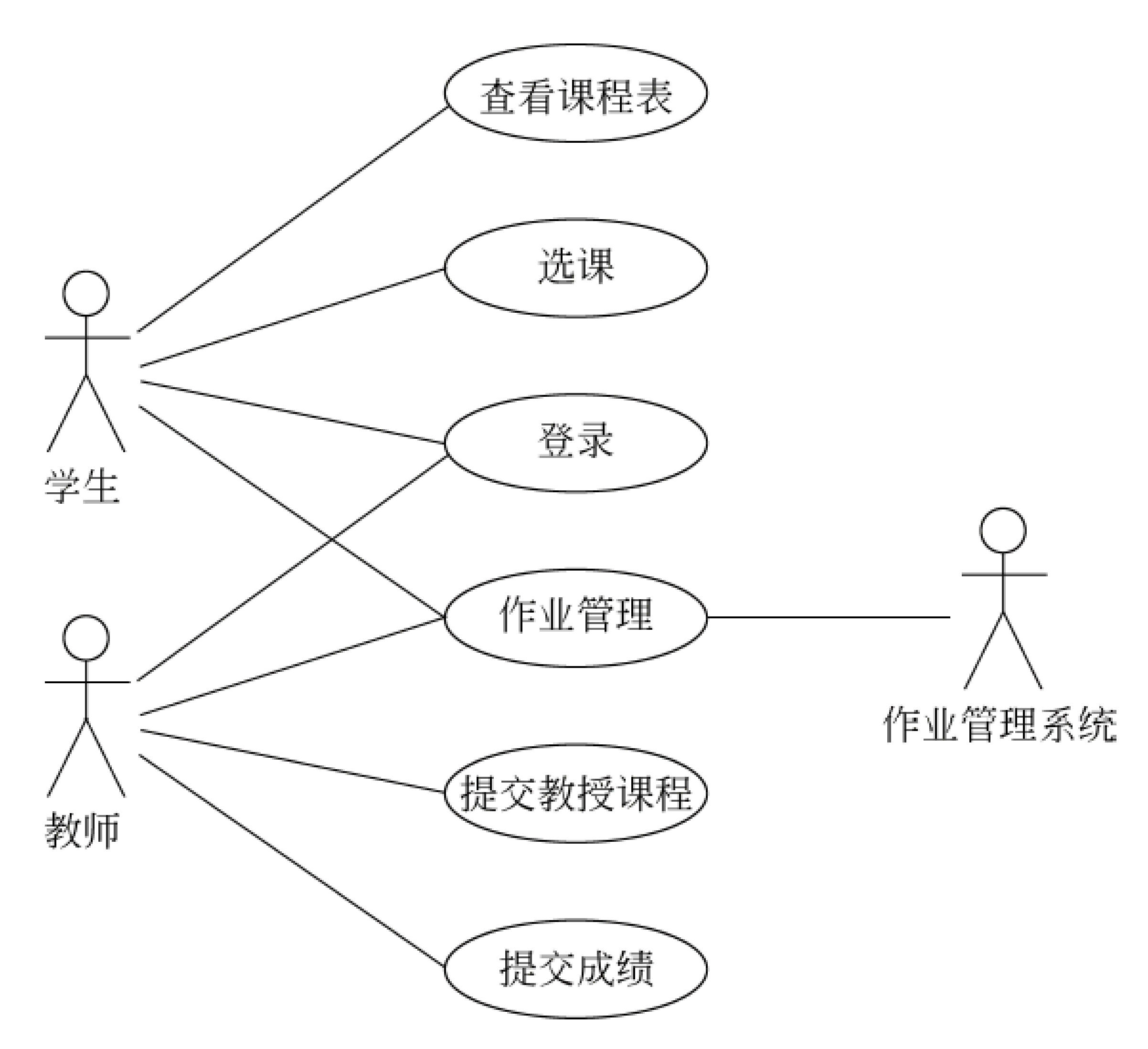


图 7-4　学生课程管理系统用例图

个执行者需要协作共同完成这个用例。因此，测试作业管理这个用例就同时测试了学生、教师和作业管理系统三个执行者，学生和教师可以不再选择其他的用例进行测试。还可以看到登录也同时与学生、教师相关联，但是登录用例不能同时测试学生和教师，因为学生和教师不是共同实现登录用例的，而是分别实现的。因此，如果学生和教师都选择登录进行测试，登录用例还是需要测试两次。表 7-2 是一种测试的用例列表(可以选择其他的用例)，后两个用例可以不进行测试，因为第一个用例已经覆盖了所有执行者。

表 7-2　学生课程管理系统测试的参与者

执　行　者	用　　例
学生、教师和作业管理系统	作业管理
学生	查看成绩单(选课/登录)
教师	登录(作业管理/提交教授课程)

2. 用例覆盖准则

用例覆盖准则要求测试用例必须覆盖用例图中所有的用例。执行者覆盖准则能够测试每个执行者，但是不能测试每个用例。用例图中的每个用例都对应一个功能，为了保证对系统功能的完整测试，每一个功能都要测试。一个用例可能会与多个执行者相关，如果每个执行者都可以独立完成该用例，则可以只选择一个执行者测试该用例。如果多个执行者需要共同协作完成，则必须所有执行者进行共同测试。

仍以学生课程管理系统为例，如图 7-4 所示学生课程管理系统用例图中共有 6 个用例，每个用例都需要测试。其中，查看课程表和选课只与学生相关，则只需要测试学生实现这两个用例。同理，需要测试教师实现提交教授课程和提交成绩单这两个用例。登录

用例同时与学生、教师相关联，并且学生和教师都可以独立完成该用例，因此只需要选择学生或者教师来测试该用例。用例作业管理同时与学生、教师和作业管理系统相关联，并且这三个执行者需要共同完成该用例，因此三个执行者必须共同测试该用例。学生课程管理系统需要测试的用例如表 7-3 所示。

表 7-3　学生课程管理系统测试的用例

执　行　者	用　　例	执　行　者	用　　例
学生	作业管理	教师	提交成绩单
学生	选课	学生（教师）	登录
教师	提交教授课程	学生、教师和作业管理系统	作业管理

3. 关系覆盖准则

关系覆盖准则要求测试用例必须覆盖用例图中所有的关系，包括泛化关系、包含关系和扩展关系。

泛化关系分为两种，一种是执行者之间的关系，另一种是用例之间的关系。执行者之间的泛化关系表示子执行者继承了父执行者的特性，又有自己独有的特性，子执行者可以执行与父执行者相关的用例。用例之间的泛化关系表示子用例是父用例的一个特殊用例。在实际测试中，子执行者不仅需要测试与自身相关的用例，还需要测试与父执行者相关的用例。如果父用例只是子用例的抽象，没有实际的功能，可以不进行测试。如果父用例有实际的内容，则仍需要进行测试。

以网购支付系统为例，一个注册用户购物后可以选择网银支付、支付宝支付和货到付款三种方式，并且如果这个注册用户是一个高级用户（具有较高的信用），则还可以进行分期付款。具体的用例图如图 7-5 所示。

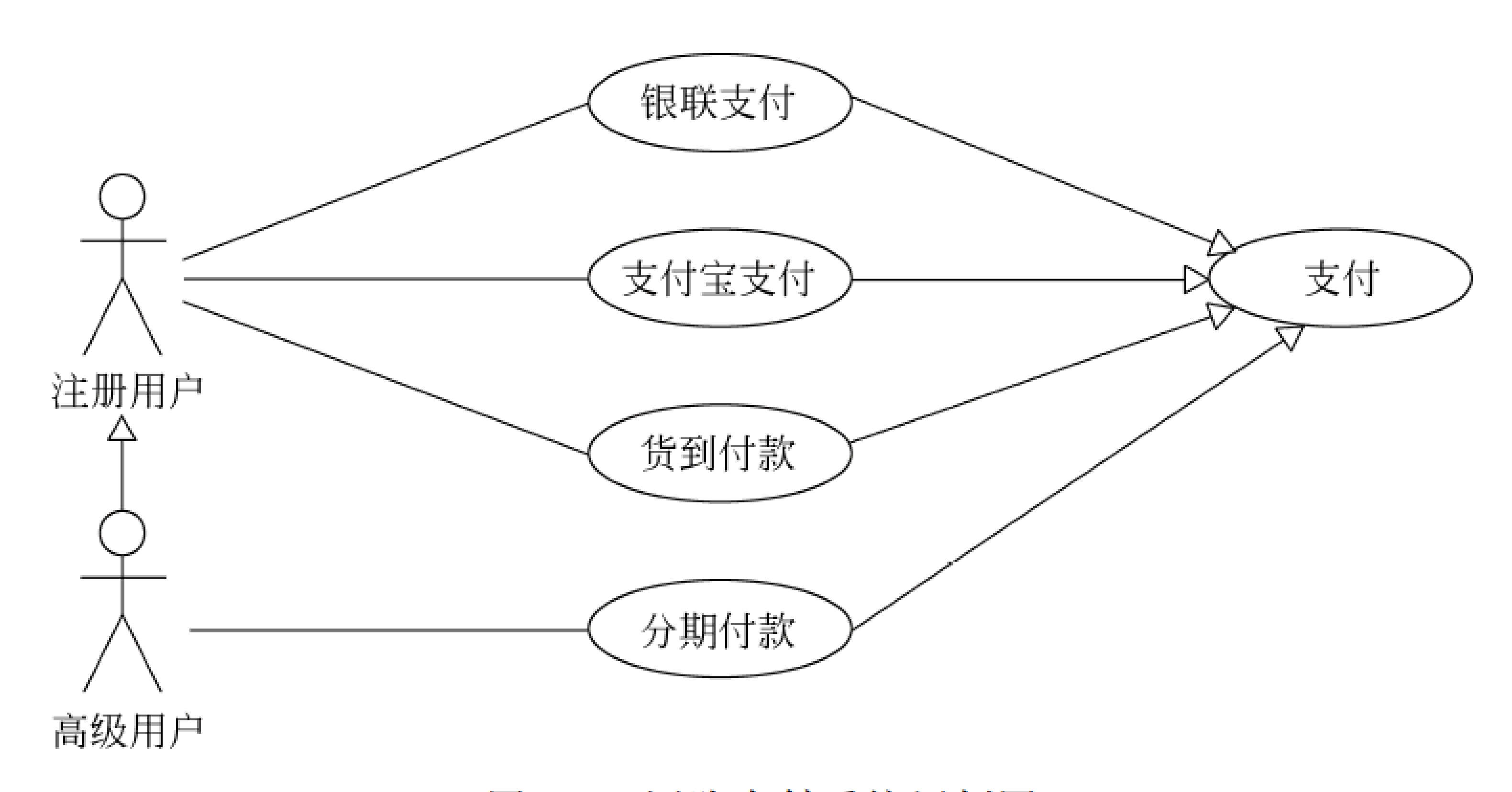


图 7-5　网购支付系统用例图

高级用户和注册用户之间是泛化关系，高级用户可以实现注册用户实现的用例，即高级用户也需要测试银联支付、支付宝支付和货到付款这三个用例。支付用例没有实际的内容，只是 4 种实际支付方式的抽象，可以不进行测试。网购支付系统需要测试的用例如

表 7-4 所示。

表 7-4 网购系统测试的用例

执行者	用　例	执行者	用　例
注册用户	银联支付	高级用户	支付宝支付
注册用户	支付宝支付	高级用户	货到支付
注册用户	货到支付	高级用户	分期支付
高级用户	银联支付		

包含关系表示一个用例包含或者使用了另一个用例中的行为。假设用例 A 包含用例 B 和用例 C，则测试用例 A 的过程中会使用用例 B 和用例 C 中的行为，也就意味着同时测试了用例 B 和用例 C。可以认为用例 B 和用例 C 是用例 A 的一部分，用例 B 和用例 C 可以不再进行单独测试。但是如果用例 B 和用例 C 独立于用例 A，直接与执行者相关联，则仍需要单独测试。

扩展关系表示一个用例扩展了另一个用例，会在一定条件下触发扩展用例。在测试的过程中，需要单独测试基础用例，也需要将扩展用例加到基础用例中进行测试。

以图书馆管理系统为例，读者可以在登录系统后修改个人信息和查阅借阅信息，也可以不用登录直接查阅图书信息。读者还可以进行借书和还书，如果还书时归还的书籍已经过期，则需要交超期罚款才能归还书籍。具体的用例图如图 7-6 所示。

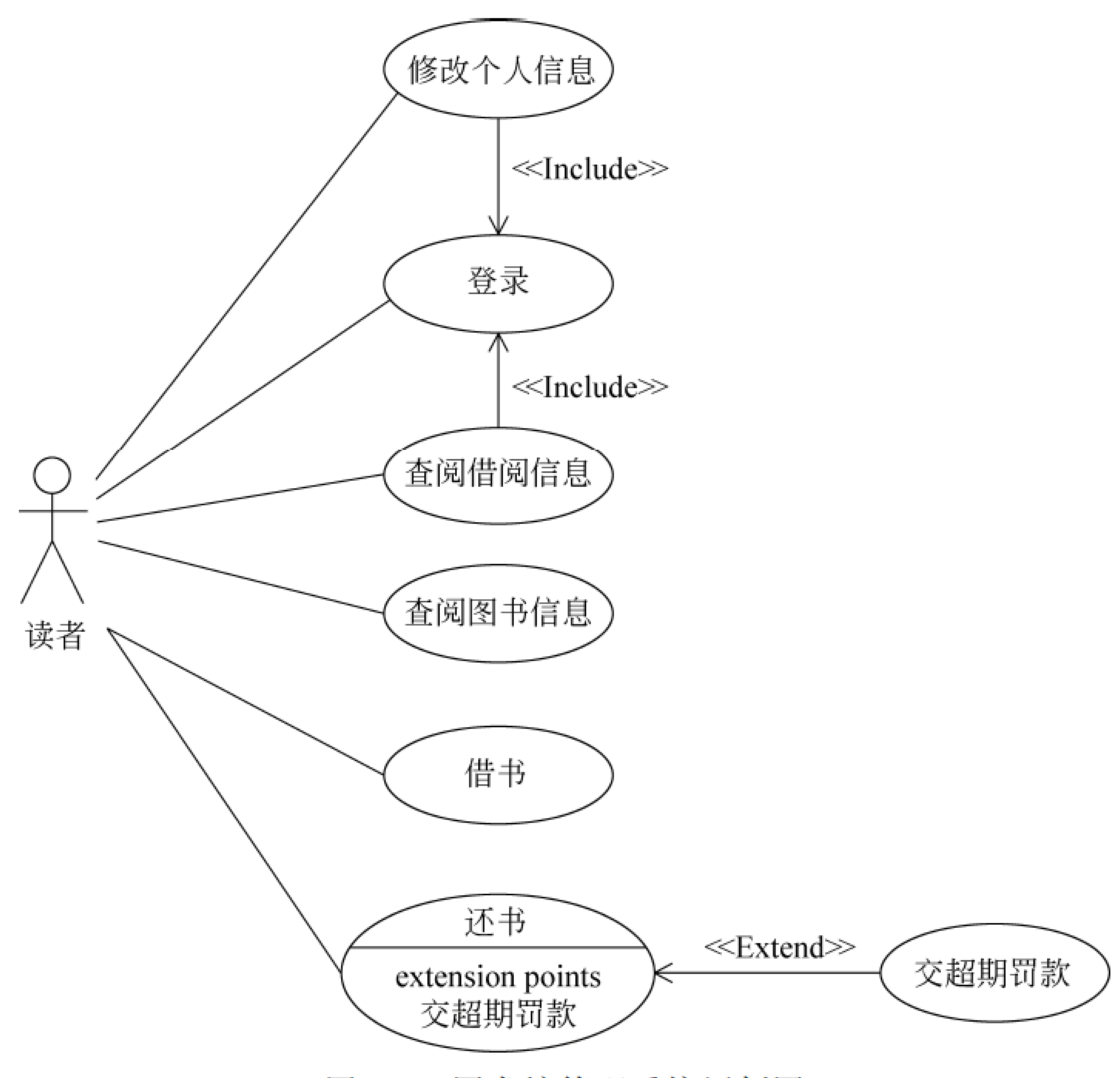


图 7-6 图书馆管理系统用例图

图 7-6 中共有 7 个用例，其中，修改个人信息和查阅借阅信息都包含登录用例，交超期罚款是还书请求的扩展用例。根据关系覆盖准则，登录用例是个人信息和查阅借阅信息的一部分，又同时与读者直接相关，是一个独立用例。因此，登录用例既要在测试修改个人信息和查阅借阅信息时进行测试，又要进行单独测试。交超期罚款是还书的一个扩展用例，根据关系覆盖准则，单独测试还书用例后，还应当将交超期罚款附加到还书用例中再进行测试。图书馆管理系统需要测试的用例如表 7-5 所示。

表 7-5 图书馆管理系统测试的用例

执行者	用　　例	执行者	用　　例
读者	修改个人信息	读者	借书
读者	查阅借阅信息	读者	还书
读者	登录	读者	还书、交超期罚款
读者	查阅图书信息		

4. 场景覆盖准则

用例图没有直接提供用例的详细信息，但可以通过用例描述来获得。用例描述对用例的细节进行了介绍，包含用例的多个属性，如表 7-6 所示。

表 7-6 用例描述模板

用例编号	用例的标志符号
用例名称	反映用例核心功能的名称
用例说明	描述用例的目标，以及用例的核心功能
执行者	执行用例的角色名称或者描述，包括人员或者与本系统有交互的其他系统
触发动作	发起该用例的执行者的动作
前置条件	在用例执行前系统期望处于的状态或者环境
后置条件	在用例执行完成以后系统期望处于的状态或者环境，包括执行成功时的状态和用例执行失败时两种情况
利益相关方	本用例所涉及的利益相关的利益体
基本路径	用例中执行的正常事件流，一般用编号表示事件执行的先后顺序
扩展路径	用例中执行的可选事件流，一般用编号表示事件执行的先后顺序
扩展点	如果本用例包括扩展或者包含其他用例，则说明扩展或者包含的其他用例，并注明使用条件
规则与约束	对该用例实现时需要考虑的业务规则、非功能需求、涉及约束等

在用例描述中，有两个重要属性：基本路径和扩展路径。基本路径表示用例中执行的正常事件流，是按照正确的事件流实现的正确流程。扩展路径是用例中执行的可选事件流，需要满足一定条件才能触发，从基本路径或者扩展路径引出。根据这两个属性可以

获得用例的场景。用例场景是指流经用例的路径，从用例的开始到用例的结束，遍历这条路径上的基本路径和扩展路径。

根据用例描述中的基本路径和扩展路径以及它们之间的关系可以得到场景图。场景图显示了基本路径和扩展路径的关系，从基本路径开始，中间会引出扩展路径，最后都是以用例的结束作为路径的终点。

下面简单举一个例子，说明场景图生成场景的过程。假设有一个基本路径和三个扩展路径，其中扩展路径 1 和扩展路径 2 都是从基本路径中引出，扩展路径 3 是从扩展路径 1 中引出。扩展路径 1 在扩展路径 2 前发生和结束，扩展路径 3 发生会导致用例的结束。根据上述描述可以得到场景图，如图 7-7 所示。

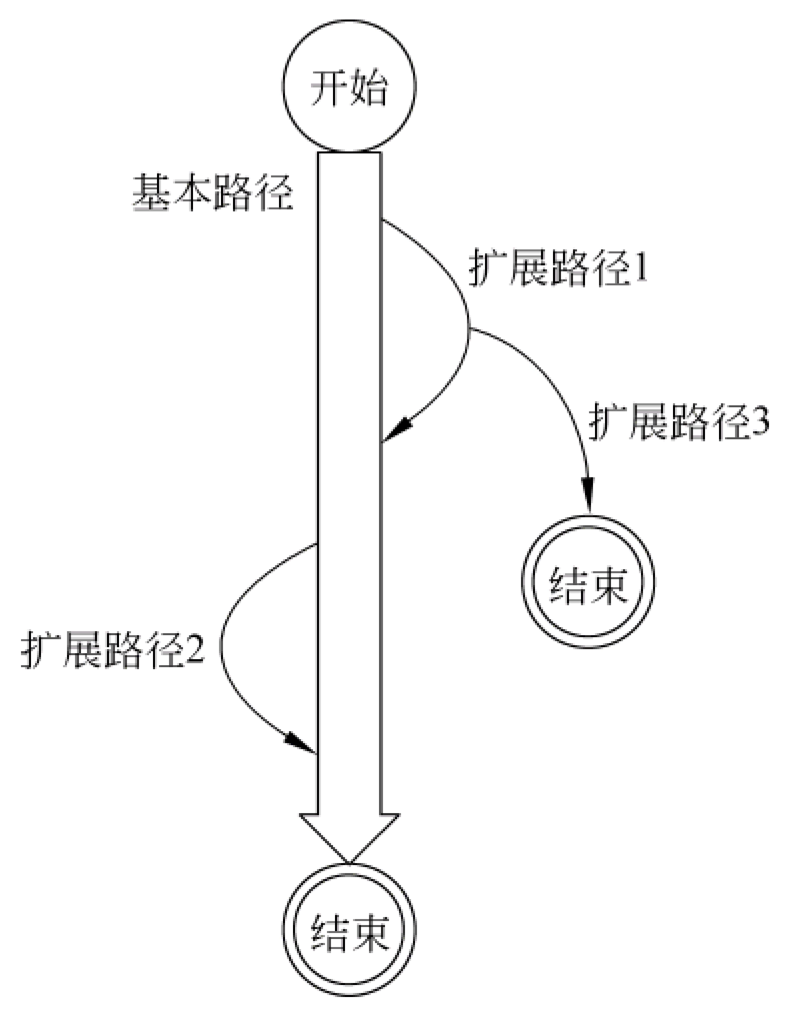


图 7-7　场景图

从基本路径开始，再结合扩展路径，可以得到以下几个场景。

场景 1：基本路径

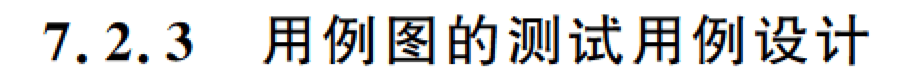
场景 2：基本路径、扩展路径 1

场景 3：基本路径、扩展路径 1、扩展路径 3

场景 4：基本路径、扩展路径 2

这 4 个场景包括从开始到结束的所有可能路径，覆盖了基本路径和扩展路径。

场景覆盖准则要求测试必须覆盖用例中所有的场景，每个场景对应一个测试用例。场景可以根据基本路径和扩展路径来获取。



7.2.3　用例图的测试用例设计

根据 UML 用例图设计测试用例，首先应当找出用例图中需要测试的用例。明确需要测试的用例后，具体的测试内容需要进一步的确定才能设计测试用例。具体测试内容的确定主要依据用例描述信息，需要满足场景覆盖准则，具体步骤如下。

(1) 根据软件规格说明书，画图得到用例图。

(2) 分析用例图的结构，识别每个执行者、用例和关系。

(3) 根据执行者覆盖准则、用例覆盖准则和关系覆盖准则，确定需要测试的用例。

(4) 对需要测试的用例进行用例描述，并根据用例描述信息(基本路径和扩展路径)生成不同的场景。

(5) 每个场景对应设计一个测试用例，整合所有测试用例形成最终测试用例集。

下面以 ATM 自动取款机为例，对用例图的测试用例设计进行详细的说明。

步骤一：根据软件规格说明书，画图得到用例图。ATM 客户取款的需求说明如下。

(1) 客户可以通过 ATM 机存款。

(2) 客户可以通过 ATM 机查询余额。

(3) 客户可以通过 ATM 机取款,同时可以查询余额。

(4) 客户可以通过 ATM 机转账,同时可以查询余额。

根据上述描述,得到用例图,如图 7-8 所示。

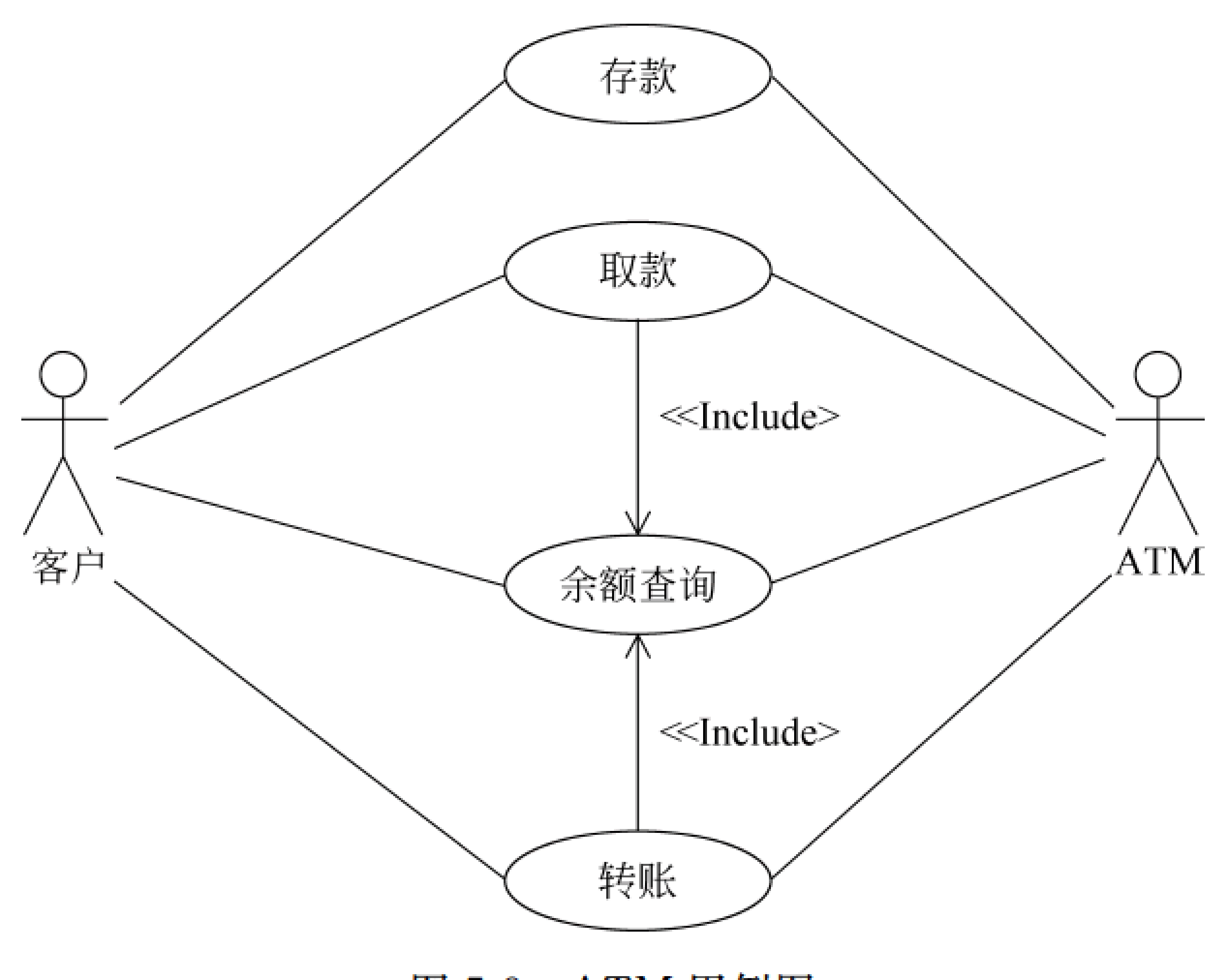


图 7-8 ATM 用例图

步骤二:分析用例图的结构,识别每个执行者、用例和关系。

从 ATM 用例图可知,有两个执行者:客户和 ATM。用例有存款、取款、余额查询和转账 4 个,且都与客户和 ATM 相关联。用例取款和转账都包含余额查询用例。

步骤三:根据执行者覆盖准则、用例覆盖准则和关系覆盖准则,确定测试的用例。

根据执行者覆盖准则和用例覆盖准则,每个执行者和用例都需要测试。客户和 ATM 共同参与执行存款、取款、余额查询和转账,这 4 个用例都要进行测试,并且只需要测试一次,因为都是客户和 ATM 协作执行的功能。

取款和转账用例都包含余额查询用例,根据关系覆盖准则,在测试取款和转账时同时测试余额查询。由于余额查询也是一个单独用例,还是需要单独测试。

总结得出:用例存款、取款、余额查询和转账都需要进行测试,并且取款和转账在测试的过程中要将余额查询作为一部分进行测试。具体需要测试的用例如表 7-7 所示。

表 7-7 ATM 自动取款机测试的用例

执行者	用 例	执行者	用 例
客户、ATM	存款	客户	余额查询
客户、ATM	取款	ATM	转账

步骤四:对需要测试的用例进行用例描述,并根据用例描述信息(基本路径和扩展路径)生成不同的场景。

这里只对取款用例进行说明,其他用例的测试可以参考此用例,将不再展开。取款用例的用例描述如表 7-8 所示。

表 7-8 取款的用例描述

用例编号	Withdraw
用例名称	取款
用例说明	输出取款金额，修改账户余额
执行者	客户、ATM
触发动作	单击"取款"
前置条件	ATM 机处于准备就绪状态
后置条件	账户余额减少，ATM 机现钞减少，客户取得现金
基本路径	1. 插入银行卡： 客户将银行卡插入 ATM 机中。 2. 验证银行卡： ATM 读取银行卡的磁条，检查是否是可接受的银行卡，得到银行卡是有效的。 3. 输入密码： 客户输入 6 位密码。 4. 验证账号和密码： ATM 验证账号和密码，账号存在且对应的密码正确。 5. 选择取款： ATM 显示操作页面，客户选择"取款"选项。 6. 输入取款金额： 客户输入取款金额。 7. 银行授权： ATM 向银行系统发送账户、密码以及取款金额，获得批准后返回更新的账户余额。 8. 提取现金： ATM 提供现金，显示取款成功。 9. 打印凭条： ATM 打印并吐出凭条，更新 ATM 内部数据。 10. 返回银行卡： ATM 询问是否继续服务，客户选择"否"，银行卡被退回，结束用例
扩展路径	1. 银行卡无效： 在基本路径的步骤 2 中，如果银行卡无效，则 ATM 返回卡，并进行通知。 2. 账户不存在： 在基本路径的步骤 4 中，如果账户不存在(账户不可用)，则 ATM 返回卡，并进行通知。 3. 密码错误： 在基本路径的步骤 4 中，如果密码错误，则客户重新输入密码。 4. 累计三次密码错误： 在扩展路径 3 中，如果密码错误三次，则 ATM 吞卡，并进行通知。 5. ATM 内没有现金： 在基本路径的步骤 5 中，如果 ATM 没有现金，则 ATM 不显示取款功能，用例终止。 6. ATM 内现金不足： 在基本路径的步骤 7 中，如果 ATM 内现金少于客户输入的取款金额，则通知客户重新输入取款金额。 7. 账户余额不足： 在基本路径的步骤 7 中，如果账户余额不足，则通知客户重新输入取款金额。 8. 超过可提取的最大提款金额： 在基本路径的步骤 7 中，如果客户输入的取款金额超过可提取的最大提款金额，则通知客户重新输入取款金额

对应。经过步骤五的分析,得到了 9 个场景,每个场景都对应不同的测试内容,得到 9 个测试用例,具体如表 7-9 所示。

表 7-9 取款的用例

用例编号	操作编号	前置条件	事件	后置条件	结果
用例 1(场景 1)	1.1	ATM 准备就绪	插入银行卡	银行卡有效	取款成功
	1.2	显示输入密码界面	输入密码	密码正确	
	1.3	显示操作界面(有“取款”选项)	选择“取款”	进入取款	
	1.4	显示取款金额输入界面	输入取款金额	ATM 提供现金	
用例 2(场景 2)	2.1	ATM 准备就绪	插入银行卡	银行卡无效	提示银行卡无效,ATM 退卡,取款失败
用例 3(场景 3)	3.1	ATM 准备就绪	插入银行卡	银行卡有效	显示账户不存在,ATM 退卡,取款失败
	3.2	显示输入密码界面	输入密码	账户不存在	
用例 4(场景 4)	4.1	ATM 准备就绪	插入银行卡	银行卡有效	取款成功
	4.2	显示输入密码界面	输入密码	密码错误	
	4.3	显示输入密码界面	输入密码	密码正确	
	4.4	显示操作界面(有“取款”选项)	选择“取款”	进入取款	
	4.5	显示取款金额输入界面	输入取款金额	ATM 提供现金	
用例 5(场景 5)	5.1	ATM 准备就绪	插入银行卡	银行卡有效	提示密码错误三次,ATM 吞卡,取款失败
	5.2	显示输入密码界面	输入密码	密码错误	
	5.3	显示输入密码界面	输入密码	密码错误	
	5.4	显示输入密码界面	输入密码	密码错误三次	
用例 6(场景 6)	6.1	ATM 准备就绪	插入银行卡	银行卡有效	没有“取款”选项,取款失败
	6.2	显示输入密码界面	输入密码	ATM 内没有现金	
用例 7(场景 7)	7.1	ATM 准备就绪	插入银行卡	银行卡有效	取款成功
	7.2	显示输入密码界面	输入密码	密码正确	
	7.3	显示操作界面(有“取款”选项)	选择“取款”	进入取款	
	7.4	显示取款金额输入界面	输入取款金额	ATM 内现金不足	
	7.5	显示取款金额输入界面	输入取款金额	ATM 提供现金	

续表

用例编号	操作编号	前置条件	事件	后置条件	结果
用例8（场景8）	8.1	ATM准备就绪	插入银行卡	银行卡有效	取款成功
	8.2	显示输入密码界面	输入密码	密码正确	
	8.3	显示操作界面（有“取款”选项）	选择“取款”	进入取款	
	8.4	显示取款金额输入界面	输入取款金额	账户余额不足	
	8.5	显示取款金额输入界面	输入取款金额	ATM提供现金	
用例9（场景9）	9.1	ATM准备就绪	插入银行卡	银行卡有效	取款成功
	9.2	显示输入密码界面	输入密码	密码正确	
	9.3	显示操作界面（有“取款”选项）	选择“取款”	进入取款	
	9.4	显示取款金额输入界面	输入取款金额	超过可提取的最大提款金额	
	9.5	显示取款金额输入界面	输入取款金额	ATM提供现金	

7.3 基于类图的软件测试

7.3.1 类图的概念

面向对象开发中的一个重点是类，类将现实中的事物进行了抽象，而对象就是类的实例。类图可以通过描述类和类之间的关系反映软件的一个系统架构，因此类图在UML建模中是很重要的。

类定义了一组属性和行为的对象。类在UML中通常用一个三格的矩形表示，分别由类名、属性和行为操作三部分构成。图7-10给出了一个UML类图的示例，类名是Employee。这个类具有三个属性，分别为companyName、name、age。其中，companyName为类属性，而name和age为对象属性，对象属性将随着对象的消失而消失。类具有4个行为操作，其中，__init__()在Python语言中，执行类的初始化。

Employee　类名

+companyName
+name
+age　属性

+showNumber()
+_init_()
+showName()
+showAge()
+showCompanyName()　行为

图7-10　一个UML类图示例

类之间的关系主要包括：泛化、关联、依赖、聚合、组合。

泛化也就是继承关系，也称为“is-a-kind-of”关系，泛化关系描述了父类与子类之间的关系。子类继承父类的功能，并可以增加它自己的新功能。在UML中，泛化关系用带空

行修改。控制类协调了借书边界类、用户信息实体类和书籍信息实体类的动作。

在类图中,类是现实生活中对象的一种抽象,这些对象具有相同的属性和操作。在表示类图时,类的主要内容有类的名称、类的属性和类的操作。类的名称用于标记一个类,从而区别于其他的类,不可省略。类的属性描述了类代表的对象的统一特征,如果没有属性,则可以省略。例如,控制类可能没有属性,只有操作。类的操作是指对数据的处理方法,一个类中的每个操作都有唯一的名称,同时可以对操作的返回值和输入参数进行说明,也可省略。

类图的另一个核心内容是类与类之间的关系,主要有依赖、关联、泛化和实现 4 种关系。具体说明如下。

1. 关联关系

关联关系描述了类与类之间的关系,一般表示一个类的对象“has a”另一个类的对象,也可以是自身的关联。关联关系主要有 4 种,如图 7-11 所示,其中,图 7-11(a)表示单向关联,图 7-11(b)表示双向关联,图 7-11(c)表示组合关系,图 7-11(d)表示聚合关系。假设一个类与另一个类关联,那么其中一个类可以直接获取另一个类的实例或者使用它。关联关系中很重要的一个特性是多重性,表示一个类的对象对应另一个类的对象的数量。常用的多重性有一对一、一对多和多对多。例如,员工管理系统中,公司和职工的关系是关联关系,一个公司有多个职工,但是一个职工只能属于一个公司,多重性用一对多表示。

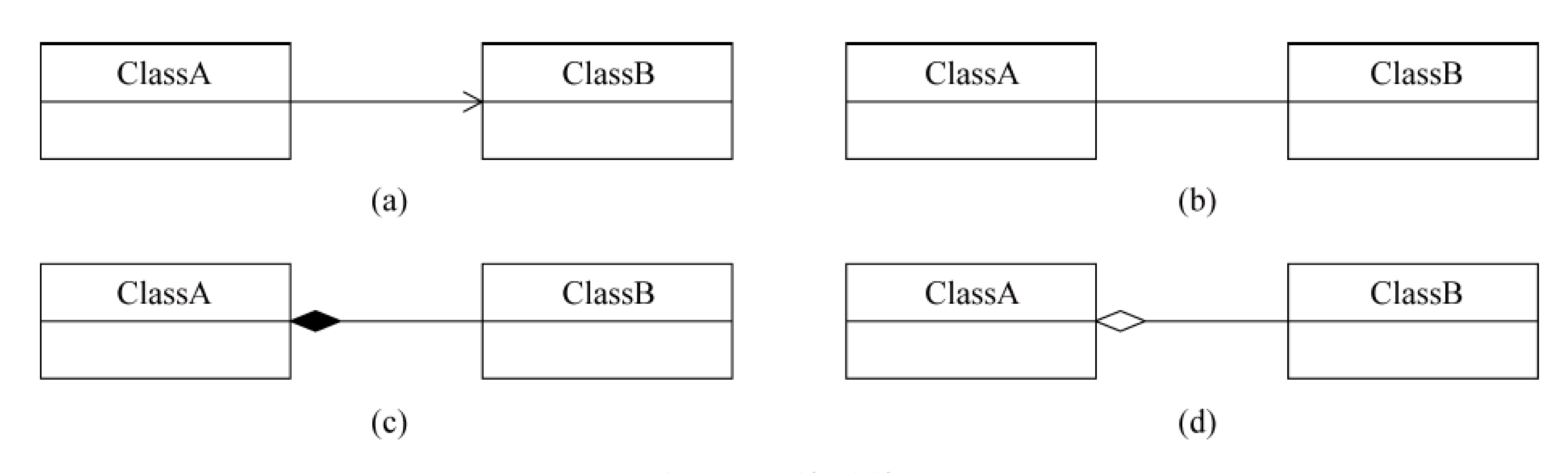


图 7-11　关联关系

2. 依赖关系

依赖关系描述了两个类之间的关系,如果一个类发生变化会引起另一个类的变化,即一个类依赖于另一个类,在 UML 类图中的表示如图 7-12 所示。例如,员工管理系统中的员工绩效考核表和员工,员工绩效考核表依赖于员工,员工发生变化,员工绩效考核表也会有相应的变化。依赖关系主要有 5 种,分别是使用依赖、抽象依赖、授权依赖、绑定依赖和替代依赖。

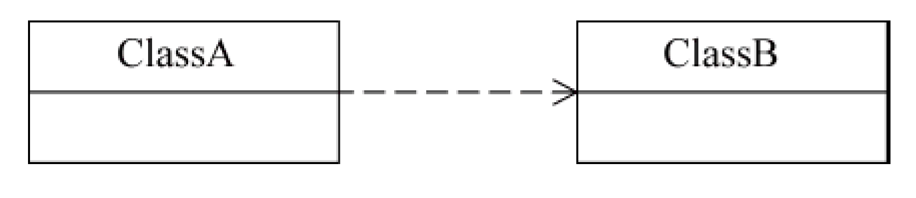


图 7-12　依赖关系

3. 泛化关系

泛化关系描述了父类和其一个或多个子类之间的关系，表示子类"is a kind of"父类，UML 类图中的表示如图 7-13 所示。父类代表所有子类的共同特性，子类代表一种个性，子类在父类的基础上进行泛化。例如，员工管理系统中的正式员工和前台服务人员，前台服务人员是一种正式员工，正式员工是父类，前台服务人员是子类。

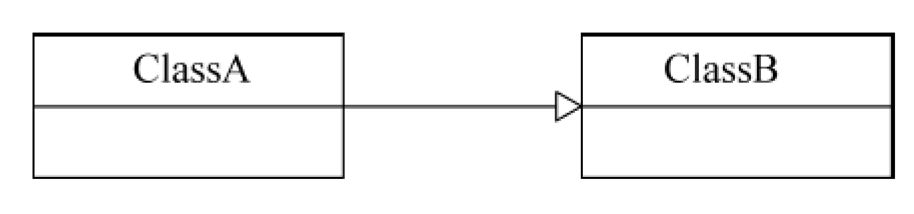


图 7-13 泛化关系

4. 实现关系

一个类可能需要实现一系列的方法，如果不把这些方法写在这个类中，可以用实现关系联系到一个接口上，表示这个类实现了这个接口，UML 类图中的实现关系如图 7-14 所示。例如，员工管理系统中，管理人员和普通员工都可以进行工资结算，但是有不同的结算方式。可以将工资结算作为一个接口，管理人员和普通员工都实现这个接口，同时进行不同的实现。在 Python 中没有接口这个概念，可以直接通过使用同名变量和方法来实现多态性，因此不用去考虑实现关系。

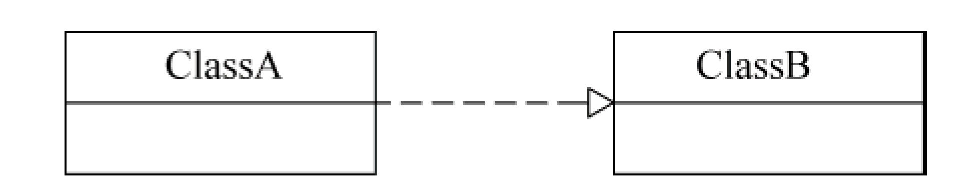


图 7-14 实现关系

类图所涉及的图标，如表 7-10 所示。

表 7-10 类图的图标表示

名 称	解 释	图示
类	表示一个类	
泛化	一个类对另一个类泛化	
实现	一个类实现另一个类	
关联	一个类的对象包含另一个类的对象	
n 元关联	多个类相互关联	
聚合	整体和部分的关系，生命周期不一致	
组合	整体和部分的关系，生命周期一致	
依赖	一个类依赖于另一个类	
协作	用于多个元素共同完成目标的容器	
注释	对元素进行解释	
约束	约束条件	

进行重写,新增了方法 operation3()和 operation4()。根据准则三,首先测试父类 A,类 B 需要测试方法 operation1()和 operation2()。类 C 需要测试方法 operation3()和 operation4()。

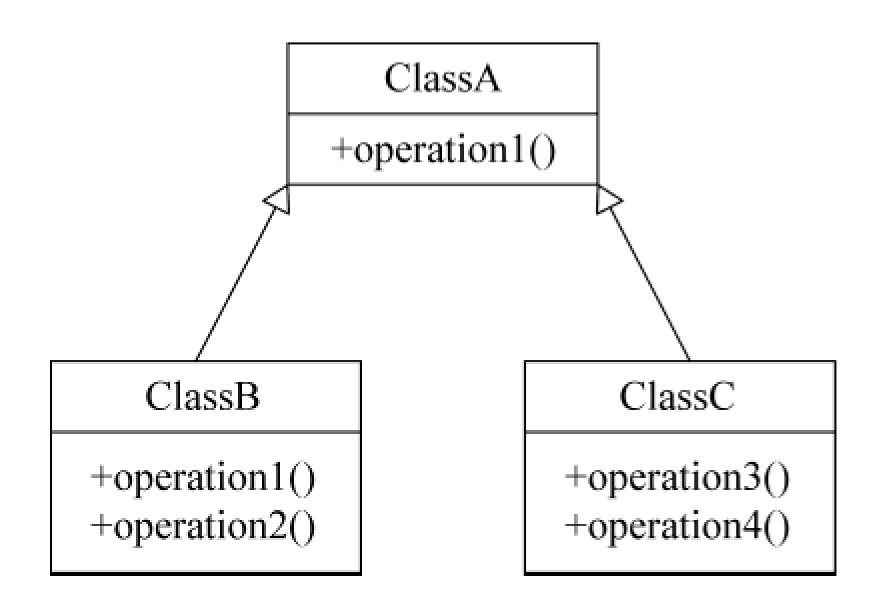


图 7-17 两个子类继承一个父类

准则四:如果存在多态性,则每一种可能的形态都需要测试一次。

多态是指发送一个消息给一个对象,该对象自行决定响应的方式,即不同对象在处理同一个消息时会有不同的行为。典型的例子是方法的重载,一个方法可以有多种实现。当出现多种形态时,为了保证测试的全面,需要将所有形态都测试一次。假设 Python 程序中有一个方法的参数为 * args,表示任何多个无名参数。在测试方法时,应当测试各种参数的情况,具体可以结合方法的具体内容展开。

准则五:如果存在组合和聚合关系,则将相关类进行组合共同测试。

组合和聚类关系表示整体和部分的关系,对整体的测试会涉及对部分的测试,有些部分可能无法单独测试。以图 7-18 为例,图中有 4 个类,其中,类 B、类 C、类 D 与类 A 存在着组合关系,类 A 是整体,类 B、类 C 和类 D 是部分,是类 A 的属性。测试的过程中对类 A 进行测试必然会涉及对类 B、类 C 和类 D 的测试,不应该拆分。

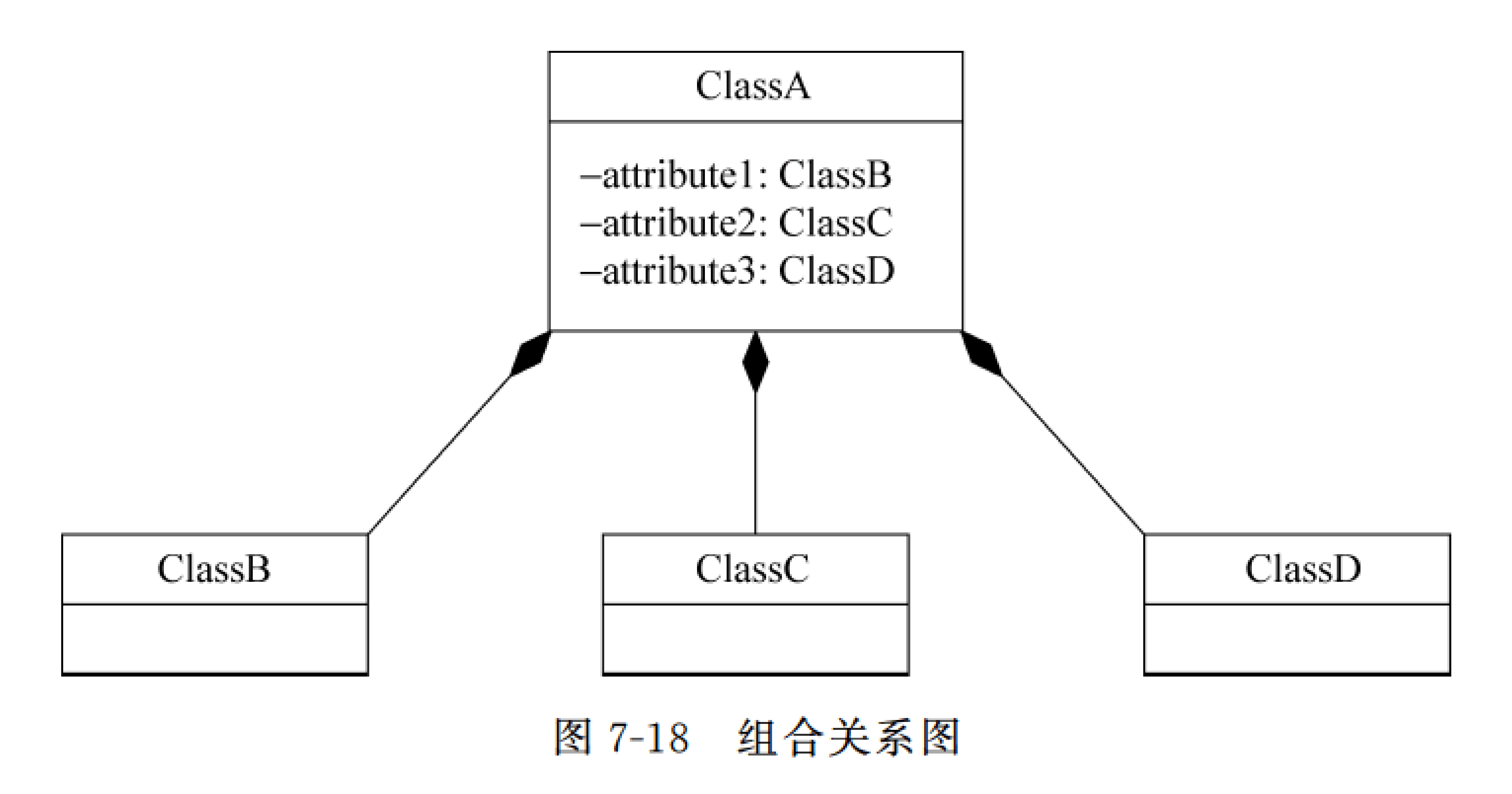


图 7-18 组合关系图

7.3.3 类图的测试用例设计

根据 UML 类图设计测试用例,需要检查类的操作和类之间的交互。根据 7.3.2 节的覆盖原则来设计测试用例,具体步骤如下。

(1) 根据软件规格说明书,画图得到类图。

(2) 分析类图的结构,识别每个类、关系和设计模式。

(3) 根据类图的覆盖准则,设计测试用例。

下面举两个简单的例子,对类图的测试用例设计进行说明。

例 7-1 生产苹果产品。

步骤一:根据生产苹果产品的软件规格说明书,画图得到类图。

续表

步骤	输　　入	预 期 输 出
4	调用 Imac1 的 surfInternet()方法	实现 Iphone 上网的功能
5	用 IpadFactory 生产 Ipad	产生一个 Ipad 的实例 Ipad1
6	调用 Ipad1 的 surfInternet()方法	实现 Ipad 上网的功能

例 7-2　翻译软件。

步骤一：根据翻译软件的软件规格说明书，画图得到类图。

翻译软件的功能是可以将英语、日语和德语翻译成中文。根据上述描述，得到类图，如图 7-20 所示。

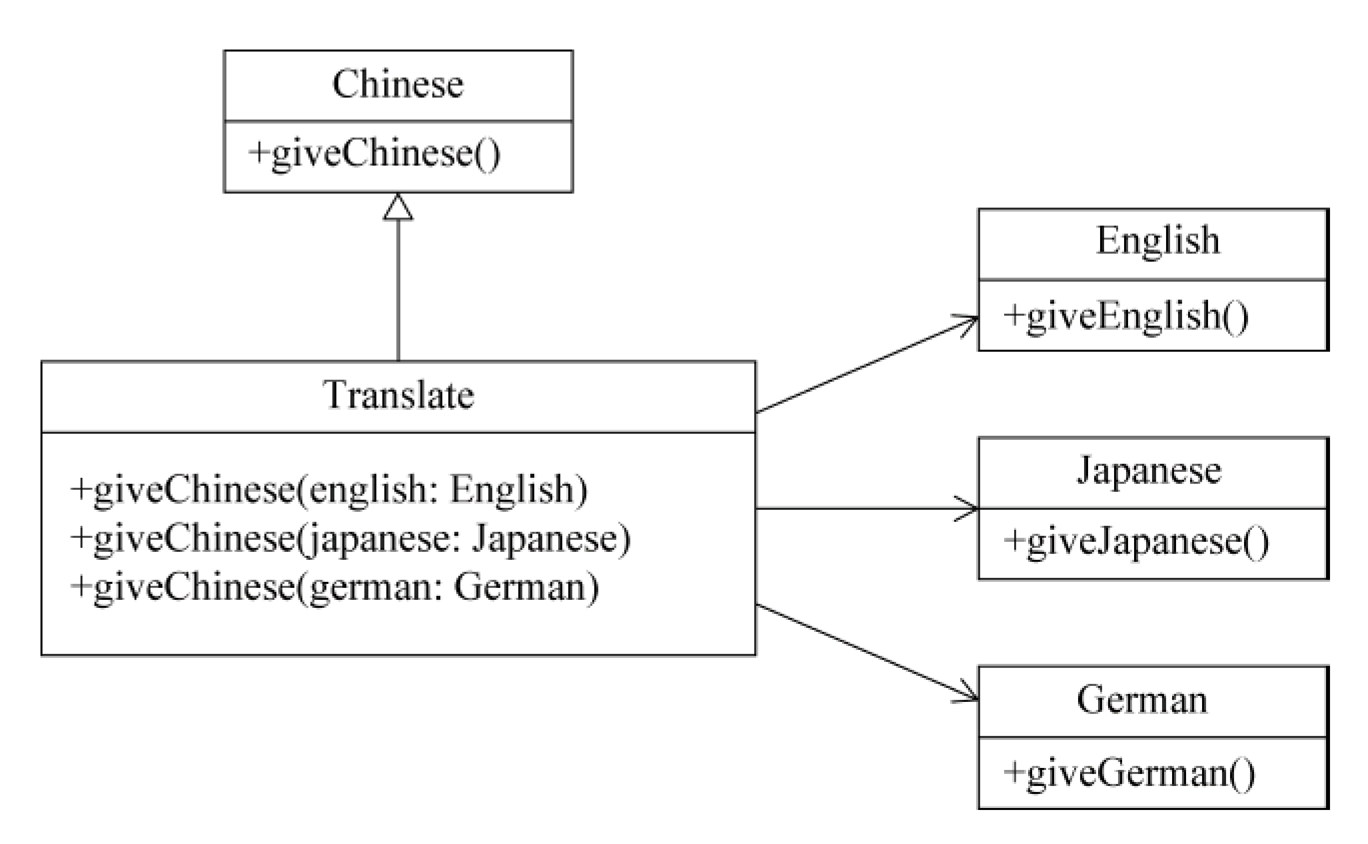


图 7-20　中文翻译

步骤二：分析类图的结构，识别图 7-20 中文翻译中的每个类、关系和设计模式。

图 7-20 中有 5 个类，使用了适配器模式。类 Translate 继承了类 Chinese，并重写了方法 giveChinese()，并且存在方法的重载。类 Translate 依赖于类 English、Japanese 和 German，giveChinese()函数的参数是这三个类。

步骤三：根据类图的覆盖准则，设计测试用例。

类 Translate 继承了类 Chinese，并且重写了父类的方法 giveChinese()，根据准则三，方法 giveChinese()需要单独进行测试。类 Translate 重载了方法 getChinese()，根据准则四，这三个方法都要测试。类 Translate 的三个方法 giveChinese()参数的类型分别是 English、Japanese 和 German，根据准则二，首先实例化类 English、Japanese 和 German，再把对象传给类 Translate 作为三个方法 giveChinese()的参数，需要和方法的参数匹配。

根据分析，可以设计测试用例，如表 7-12 所示。

表 7-12　翻译软件的测试用例

步骤	输　　入	预 期 输 出
1	实例化 English	产生一个 English 的实例 English1
2	实例化 Japanese	产生一个 Japanese 的实例 Japanese1

(2) 对象流：连接活动与数据或者动作与数据的边。通常用对象表示动作或者动作的输入和输出，常见的是输入和输出数据。

对活动图中的各元素进行形式化定义，方便识别与操作，具体定义如下。

(1) N 表示活动图中所有活动节点的非空有限集合，活动节点包括执行节点、控制节点和对象节点，$N=N_B \cup N_C \cup N_O$。

N_B 表示活动图中的所有执行节点的非空有限集合，表示如下：

$$N_B=\{N_1,N_2,\cdots,N_i,\cdots,N_n\}$$

其中，N_i 表示一个执行节点，标号为 i。

N_C 表示活动图中的所有控制节点的非空有限集合，表示如下：

$$N_C=\{C_1,C_2,\cdots,C_i,\cdots,C_n\}$$

其中，C_i 表示一个控制节点，标号为 i。其中，开始节点用 initial 表示，终止节点用 final 表示。

N_O 表示活动图中的所有对象节点的非空有限集合，表示如下：

$$N_O=\{O_1,O_2,\cdots,O_i,\cdots,O_n\}$$

其中，O_i 表示一个对象节点，标号为 i。

(2) E 表示活动图中所有活动边的非空有限结合，表示如下：

$$E=\{E_1,E_2,\cdots,E_i,\cdots,E_n\}$$

其中，E_i 表示一条活动边，标号为 i。

活动图所涉及的图标，如表 7-13 所示。

表 7-13　活动图的图标表示

名　　称	解　　释	图示
开始节点	整个活动的开始	
终止节点	整个活动的结束	
活动节点	一个活动	
接收事件动作	接收信号，执行相应动作	
发送信号动作	发送信号，触发另一个动作	
活动参数节点	接收一个活动的输入和提供一个活动的输出	
决策节点	判断执行多个动作	
合并节点	判断执行一个动作	
分叉节点	将一个流变为多个流	
汇合节点	将多个流变为一个流	
对象节点	一个分类的实例	
控制流	一个活动结束后指向另一个活动	CF
对象流	活动或动作连接对象或数据	OF
异常处理程序	指向活动中的异常处理	

动边覆盖准则设计的测试用例不一定能够覆盖所有的路径。

以图 7-22 为例，该图中 N4 既是前两个分支的合并节点，又是后两个分支的决策节点。

根据活动边覆盖准则，两条路径可以覆盖所有的活动边，分别如下。

路径 1：initial-N1-N2-N4-N6-final

路径 2：initial-N1-N3-N4-N5-final

可以发现，上述两条路径没有覆盖所有的路径。路径 initial-N1-N2-N4-N5-final 和路径 initial-N1-N3-N4-N6-final 这两条路径没有被覆盖。

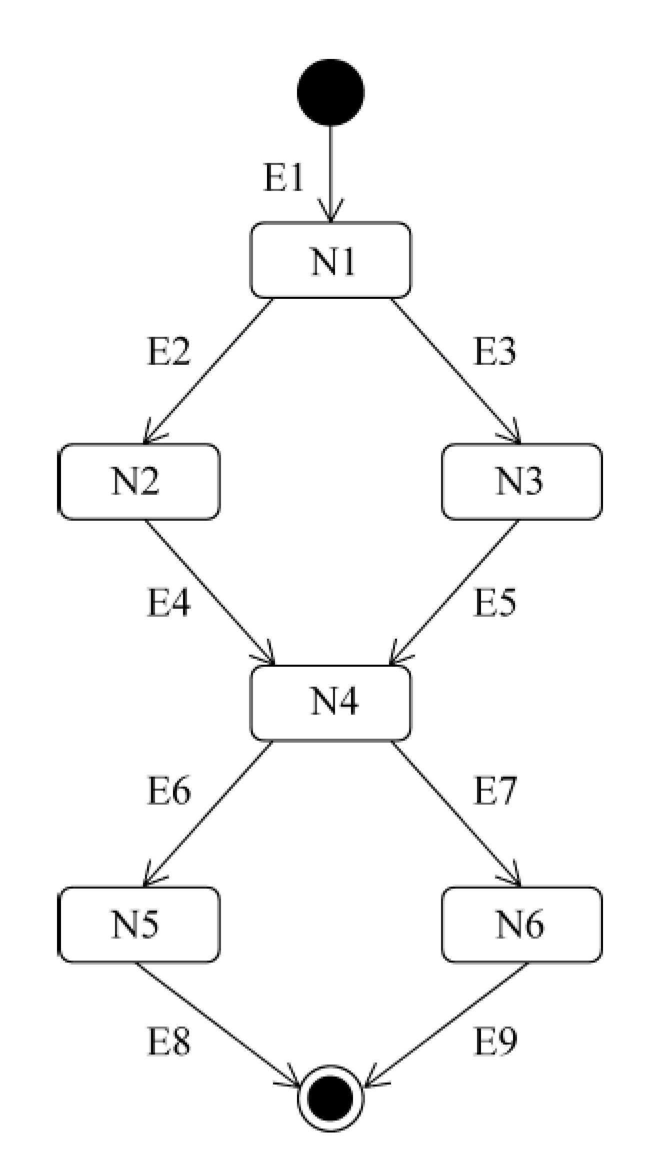


图 7-22 合并节点和决策节点是同一个节点

3. 路径覆盖准则

路径覆盖准则要求覆盖从开始节点到终止节点的所有路径，即所有路径都遍历一遍。每个活动或者活动边至少在一个路径中出现。根据路径覆盖准则，图 7-22 可得到如下 4 条路径。

路径 1：initial-N1-N2-N4-N6-final

路径 2：initial-N1-N3-N4-N5-final

路径 3：initial-N1-N2-N4-N5-final

路径 4：initial-N1-N3-N4-N6-final

路径覆盖准则弥补了活动边覆盖准则的不足之处，可以更加全面地对各种情况进行测试。在活动图中，可能存在并发结构。下面在有并发结构的活动图中，详细分析路径选择的方法。

并发结构是活动图中很常见的一种结构，通常有几个活动是可以同时执行的，而不会相互影响，如图 7-23 所示。从 N1 出发的控制流发生了分叉，最后又汇合到 N6，一个控制流被分为两个并发运行的分支。

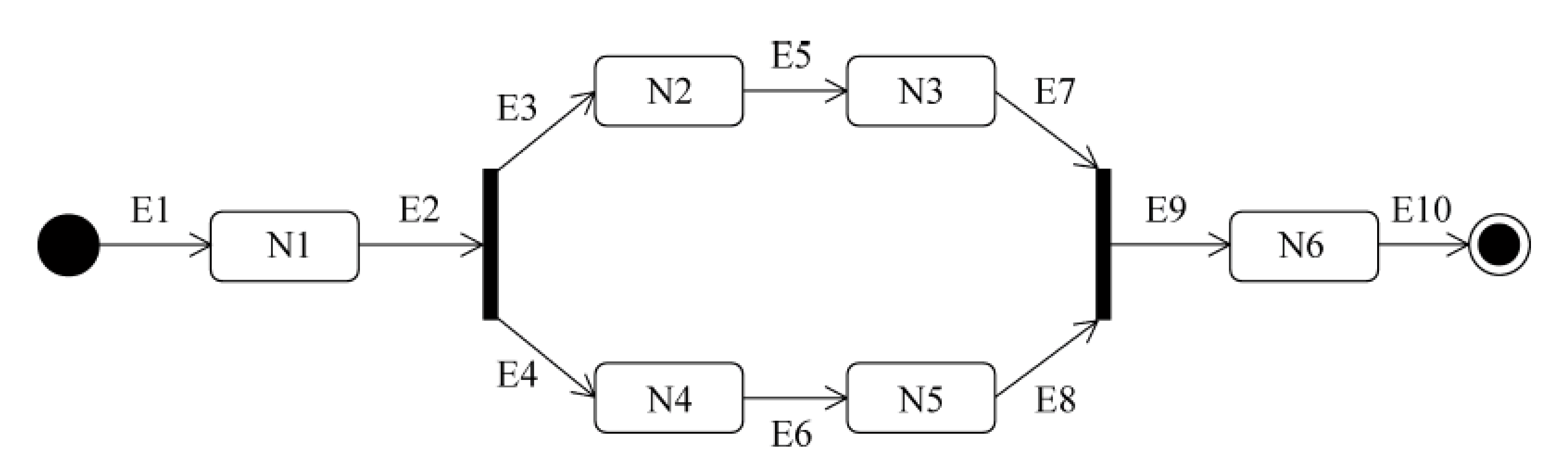


图 7-23 含有并发结构的活动图

活动图中存在并发结构，涉及偏序关系的分析。偏序关系 $N_i < N_j$ 表示活动 N_i 会在 N_j 之前执行。也就是认为，N_j 发生的条件之一是 N_i 已经发生。并发结构中的偏序关系确定有以下三种情况。

(1) 如果 N_i 是分叉节点前最后一个活动,N_j 是分叉节点后并发分支的第一个活动,则 $N_i < N_j$。

(2) 如果 N_i 是汇合节点后第一个活动,N_j 是汇合节点前并发分支的最后一个活动,则 $N_j < N_i$。

(3) 如果 N_i 和 N_j 是一个并发分支的两个活动,其中 N_i 在 N_j 的前面,则 $N_i < N_j$。

根据上述对偏序关系的描述,分析图 7-23 的偏序关系,可以得到以下 6 对偏序关系:N1<N2,N1<N4,N2<N3,N4<N5,N3<N6,N5<N6。

偏序关系确定了路径的活动是有执行的顺序的,但分支之间活动的执行是没有顺序的。例如,在执行 N1 时,可以同时执行 N4 或者 N5。分支上的活动节点必须全部执行,但顺序可以进行调整,得到以下 6 条路径。

路径 1:initial-N1-N2-N4-N3-N5-N6-final

路径 2:initial-N1-N2-N4-N5-N3-N6-final

路径 3:initial-N1-N2-N3-N4-N5-N6-final

路径 4:initial-N1-N4-N2-N5-N3-N6-final

路径 5:initial-N1-N4-N2-N3-N5-N6-final

路径 6:initial-N1-N4-N5-N2-N3-N6-final

可以发现,N1 和 N6 的位置是不变的,在满足偏序关系的前提下,可以调整 N2、N3、N4 和 N5 的顺序。

4. 分支覆盖准则

分支存在于有判断的情况下,覆盖分支要遍历所有分支至少一遍。判断分支是控制流在经过判断节点后发生了分流而产生的分支。当满足一定条件时会执行相关的分支。如图 7-24 所示,N1 执行完毕后通过 C1 判断,产生了三个分支。

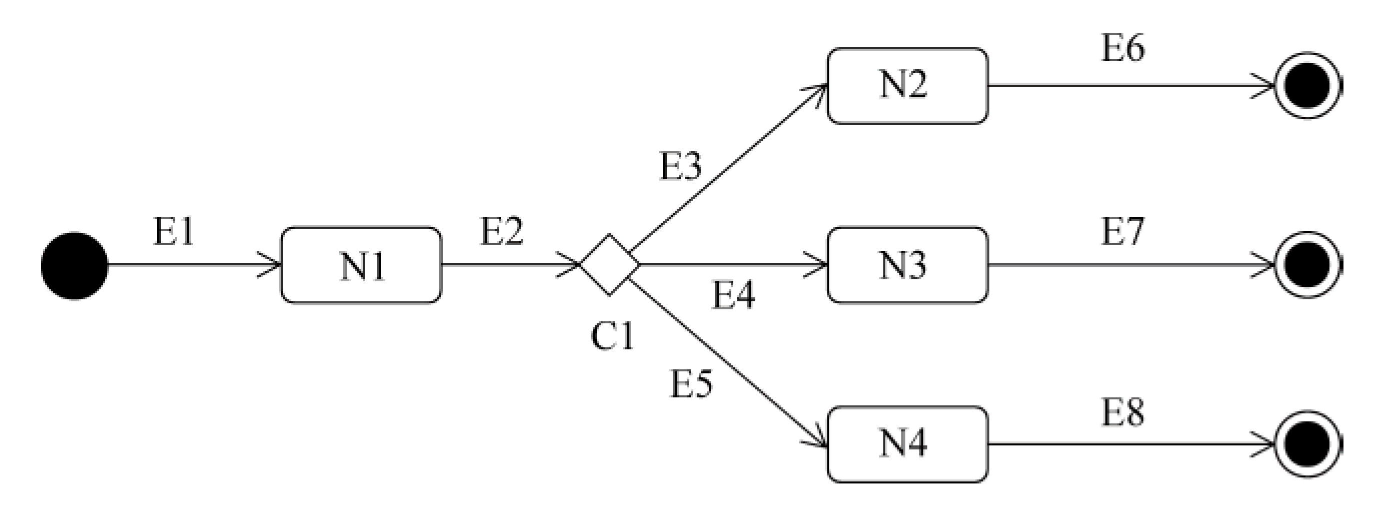


图 7-24 含有判断分支的活动图

实际测试中,所有判断分支都需要进行测试,满足分支覆盖准则。在图 7-24 中,总共有以下三条分支。

分支 1:initial-C1-N2-final

分支 2:initial-C1-N3-final

分支 3:initial-C1-N4-final

分支覆盖准则的要求是每条分支都要执行一次,同时分支之间是不能同时执行的。

5. 循环覆盖准则

循环覆盖准则要求覆盖活动图中所有的循环，循环的次数为 0 次、1 次和 n 次。n 的选取可以根据实际情况来确定。循环的发生需要满足一定的条件，会导致一些活动重复地执行。但因为循环可能是无穷次，会使测试陷入死循环，因此需要设定循环的次数。既要考虑不进入循环的情况，同时又要考虑进入循环的情况，限定循环次数，减少测试负担。

循环覆盖准则是有必要的，循环在活动图中很多见，很可能存在缺陷，必须要覆盖。

下面以图 7-25 为例，使用循环覆盖准则设计测试用例。在图中有一个循环，并且触发节点是 C1。

根据循环覆盖准则，分别选取覆盖循环 0 次和 1 次。可以设计以下两条路径。

路径 1：initial-N1-N2-N3-C1-N4-final

路径 2：initial-N1-N2-N3-C1-N5-N2-N3-C1-N4-final

路径 1 覆盖了循环 0 次，路径 2 覆盖了循环 1 次。这是最普通的循环，只有一个循环并且循环内部没有嵌套其他循环。实际测试中存在循环嵌套的活动图，查找路径会变得比较复杂。只有一层循环的路径只需考虑循环的次数，但是如果有多层循环就会产生很复杂的路径。

以图 7-26 为例，该活动图中包含两个循环，其中一个循环嵌套了另一个循环，循环 N1-N2-C1-N3-C2-N5-N1 内部嵌套了循环 N1-N2-C1-N4-N1。设定循环覆盖准则中的循环次数为 0 次和 1 次，那么图中的两个循环也都需要执行 0 次和 1 次。如果是没有嵌套关系的两个循环，相互之间独立，则路径的总数至少为 $2\times2=4$。如果是嵌套的两个循环，会因为循环的执行顺序不同产生不同的路径，需要更多路径才能覆盖循环。下面对测试情况进行分析。

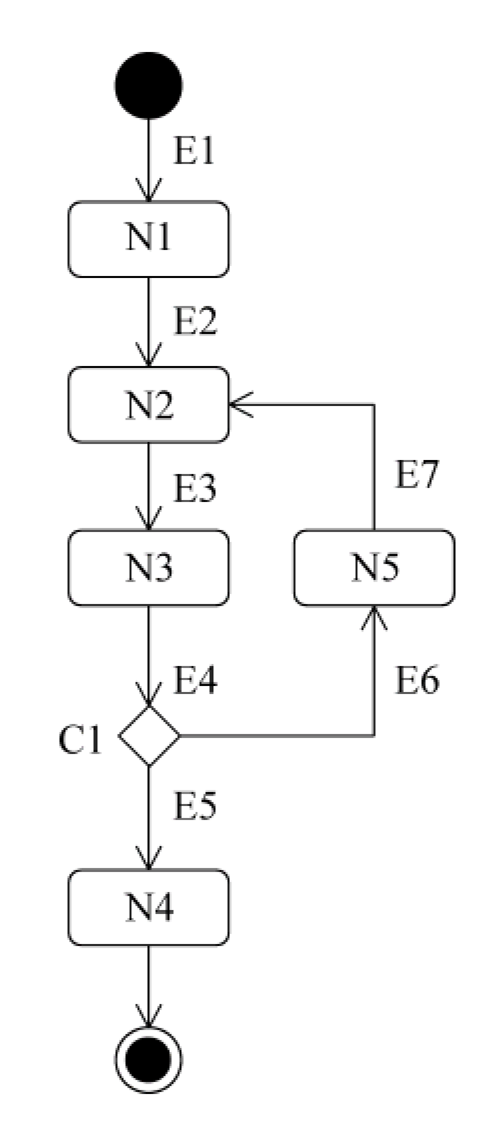


图 7-25　有一个循环的活动图

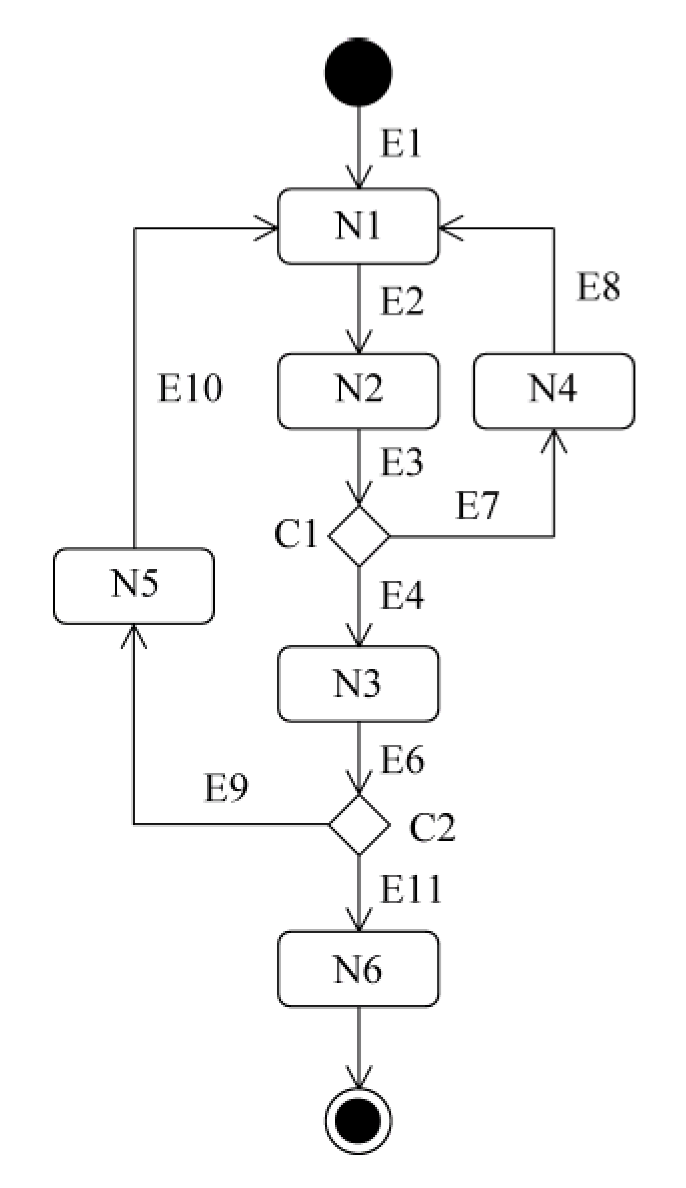


图 7-26　有循环嵌套的活动图

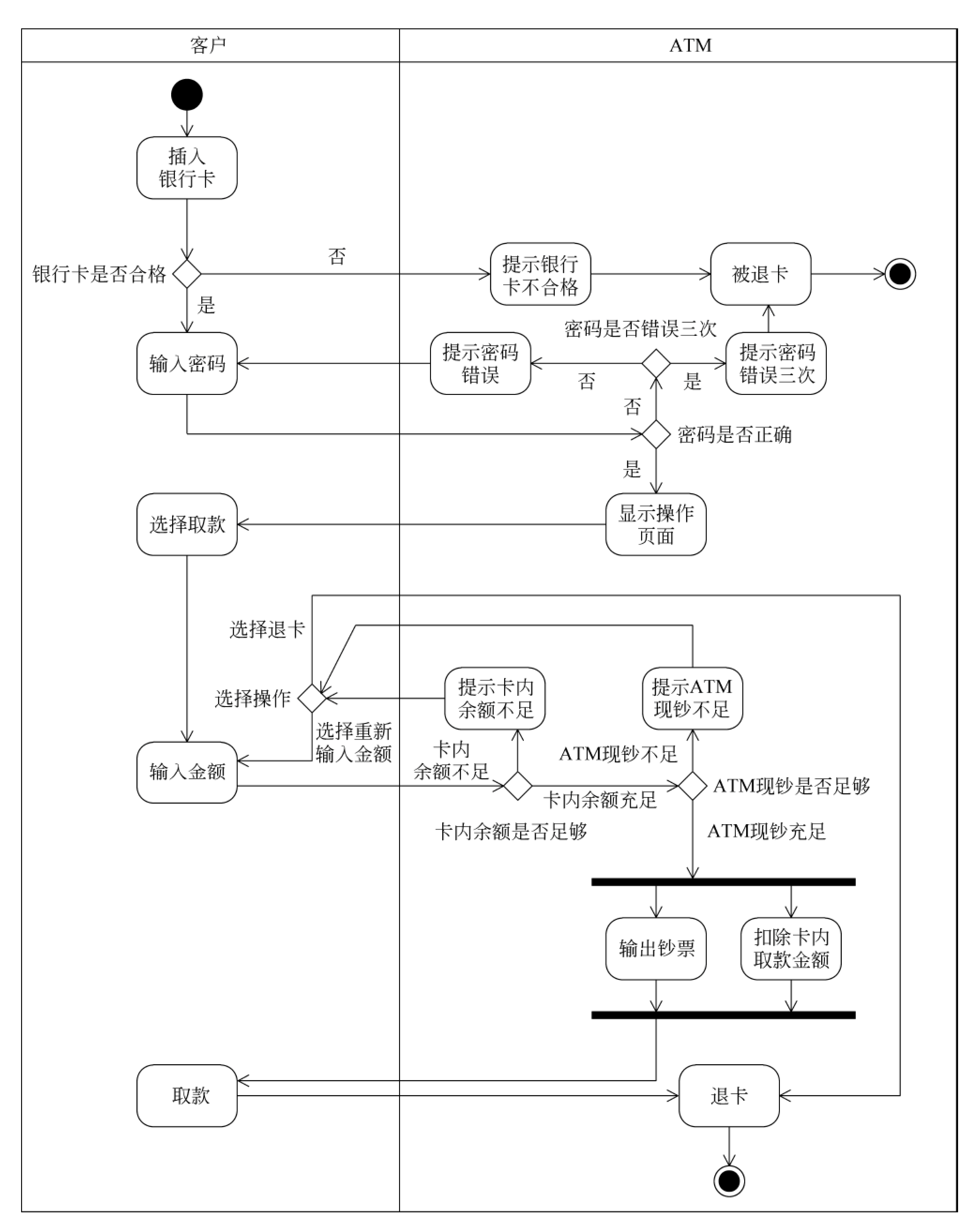


图 7-27　客户存款活动图

N10：提示卡内余额不足

N11：提示 ATM 现钞不足

N12：输出钞票

N13：扣除卡内取款金额

续表

用例编号	测试路径	条　　件
4	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→选择退卡→退卡→结束	[银行卡合格] [密码正确] [卡内余额不足] [选择退卡]
5	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→退卡→结束	[银行卡合格] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择退卡]
6	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→输出钞票→扣除卡内取款金额→取款→退卡→结束	[银行卡合格] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞充足]
7	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示卡内余额不足→选择退卡→退卡→结束	[银行卡合格] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额不足] [选择退卡]
8	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示 ATM 现钞不足→选择退卡→退卡→结束	[银行卡合格] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择退卡]
9	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→输出钞票→扣除卡内取款金额→取款→退卡→结束	[银行卡合格] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞充足]
10	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→提示卡内余额不足→选择退卡→退卡→结束	[银行卡合格] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额不足] [选择退卡]

续表

用例编号	测 试 路 径	条　　件
11	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→提示 ATM 现钞不足→选择退卡→退卡→结束	[银行卡合格] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择退卡]
12	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示 ATM 现钞不足→输入金额→输出钞票→扣除卡内取款金额→取款→退卡→结束	[银行卡合格] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞充足]
13	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示 ATM 现钞不足→输入金额→提示卡内余额不足→选择退卡→退卡→结束	[银行卡合格] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额不足] [选择退卡]
14	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示 ATM 现钞不足→输入金额→提示 ATM 现钞不足→选择退卡→退卡→结束	[银行卡合格] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择退卡]
15	开始→插入银行卡→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→提示卡内余额不足→输入金额→输出钞票→扣除卡内取款金额→取款→退卡→结束	[银行卡合格] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞充足]

续表

用例编号	测试路径	条件
22	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示卡内余额不足→选择退卡→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额不足] [选择退卡]
23	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示 ATM 现钞不足→选择退卡→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择退卡]
24	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→输出钞票→扣除卡内取款金额→取款→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞充足]
25	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→提示卡内余额不足→选择退卡→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额不足] [选择退卡]
26	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→提示 ATM 现钞不足→选择退卡→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择退卡]

续表

用例编号	测试路径	条　件
27	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示 ATM 现钞不足→输入金额→输出钞票→扣除卡内取款金额→取款→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞充足]
28	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示 ATM 现钞不足→输入金额→提示卡内余额不足→选择退卡→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额不足] [选择退卡]
29	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示卡内余额不足→输入金额→提示 ATM 现钞不足→输入金额→提示 ATM 现钞不足→选择退卡→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择退卡]
30	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→提示卡内余额不足→输入金额→输出钞票→扣除卡内取款金额→取款→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞充足]

续表

用例编号	测试路径	条件
31	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→提示卡内余额不足→输入金额→提示卡内余额不足→选择退卡→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额不足] [选择重新输入金额] [卡内余额不足] [选择退卡]
32	开始→插入银行卡→输入密码→提示密码错误→输入密码→显示操作页面→选择取款→输入金额→提示 ATM 现钞不足→输入金额→提示卡内余额不足→输入金额→提示 ATM 现钞不足→选择退卡→退卡→结束	[银行卡合格] [密码错误] [密码正确] [卡内余额充足] [ATM 现钞不足] [选择重新输入金额] [卡内余额不足] [选择重新输入金额] [卡内余额充足] [ATM 现钞不足] [选择退卡]

步骤四：根据路径以及需要满足的条件，设计具体的测试用例。

以测试用例 18 为例，假设现有的银行卡号为 0001，密码为 123456，卡内余额为 1000，ATM 内余额为 500。则设计的测试用例如表 7-15 所示。

表 7-15 测试用例 18

步骤	输入	预期输出
1	卡号 0001	显示输入密码
2	密码 129433	提示密码输入错误
3	密码 123456	显示操作页面
4	选择取款	显示金额输入页面
5	输入金额为 300	输出钞票、扣除卡内金额、退卡

7.5 基于序列图的软件测试

7.5.1 序列图的概念

序列图，也可称为顺序图，是一种交互图，用于描述对象之间有顺序的交互，强调对象

之间消息传递的时间顺序性。时间顺序性用时间轴表示。序列图中的对象包括系统的参与者和系统中其他的对象。通常情况下，序列图用于对用例图进行建模，可以对用例行为进行描述。它可以对用例图进行补充，显示用例内部的运行，更好地体现系统的功能。

序列图是一个二维图，纵轴表示时间，即对象的生命线，横轴表示系统中的对象。在二维图中，对象之间有消息的交互，并且生命线会延伸到最后的消息交互。

序列图中有5个重要的组成部分：对象、生命线、激活、消息和交互框。

1. 对象

对象并不是指系统中类实例化后的对象，而表示系统中的参与者(角色)、类或组件。这些对象之间可以进行消息的传递。对象并不一定从一开始就存在，也可能会在过程中被创建。既可以在过程中创建对象，也可以在过程中删除对象，被删除的对象一般针对新创建的对象。发送一个删除的消息到需要删除的对象，这个对象就终止了生命周期，生命线不再延伸。

2. 生命线

生命线表示对象的生命周期，在序列图中用一条虚线或矩形长条表示。生命线显示了对象的存在时间，表示一个时间轴。

3. 激活

生命线有两种状态，分别是休眠状态和激活状态。休眠状态描述了对象仍然存在，但是没有进行消息传递，用一条虚线表示。激活状态表示对象接收或者发送消息，正在进行对象之间的交互，用一个矩形长条表示。当消息传递完毕后，对象又会回到休眠状态，等待下一次的交互。

4. 消息

消息描述了对象间交互的信息，当对象处于激活状态时，会有接收消息或者发送消息的动作。消息是两个对象之间的单向通信，在描述消息时可以添加消息的序号，对消息的顺序进行补充。消息的传递可以使用参数和返回值，更好地描述时间的内容。消息可以分为4种：同步消息、异步消息、简单消息和返回消息。同步消息是指同步进行的消息传递，下一个事件发生前必须获取返回消息。异步消息表示下一个事件发生不需要等待返回消息。简单消息没有同步和异步的区分。返回消息是指当一个消息由一个对象发给另一个对象后，会有一个结果的返回或者一个确认消息的发送。

5. 交互框

交互框是指序列图中的组合片段，是一系列消息的组合，表示一个特殊的工作流。UML序列中有多种交互框，比较常用的是备选组合片段和循环组合片段。备选组合片段在不同条件情况下，执行不同的控制流，使用alt标志。选项组合片段相当于程序中的if语句，但是没有else。如果满足if的条件，就可以执行if中的消息，只有一个控制流，符

号用 opt 表示。循环组合片段表示循环执行的控制流,使用 loop 标志。

对序列图中的各个元素进行形式化定义,方便识别与操作,具体定义如下。

(1) O 表示序列图中所有对象的非空有限集合,表示如下:

$$O=\{O_1,O_2,\cdots,O_i,\cdots,O_n\}$$

其中,O_i 表示一个消息,标号为 i。

(2) M 表示序列图中的所有消息的非空有限集合,表示如下:

$$M=\{M_1,M_2,\cdots,M_i,\cdots,M_n\}$$

其中,M_i 表示一个消息,标号为 i。

(3) F 表示序列图中的所有交互框的非空有限集合,表示如下:

$$F=\{F_1,F_2,\cdots,F_i,\cdots,F_n\}$$

其中,F_i 表示一个交互框,标号为 i。

序列图所涉及的图标,如表 7-16 所示。

表 7-16 序列图的图标表示

名　称	解　释	图示
生命线	一个对象在图中的生命周期	
边界生命线	边界类对象的生命线	
控制生命线	控制类对象的生命线	
实体生命线	实体类对象的生命线	
消息	定义了生命线之间的通信	→
返回消息	回复前一个消息传回的消息	<···
呼叫消息	对目标生命线的方法调用	
反身消息	对象与自身交互	
递归消息	反身消息的一种,可以反复调用	
折返消息	指向一个对象激活点的顶部	
来源不明的消息	消息来源未知	
目的不明的消息	消息去向未知	
参与者	系统的参与者,作为对象	
备选组合片段	同一时刻可选的消息传递	
循环组合片段	一个时刻内执行多次的交互	
框架	以组合形式描述多个交互	
延续	描述备选组合片段的分支延续	
入口	连接框架内和框架外消息的连接点	
期间限制	在时间间隔之间的限制	
时间限制	对于时间限制的说明	
注释	对元素进行解释	

7.5.2 序列图的覆盖准则

UML 序列图描述了对象之间的交互，具有时间顺序性。序列图的时间顺序性要求对象操作必须遵守特定的顺序，该顺序可以从序列图中解析获得。序列图的一条消息通常表示对象的一个操作或者引起状态改变的一个触发事件。序列图可以对用例图中的用例进行描述，使用序列图设计测试用例，可以很好地测试一个用例(功能)。

为了保证测试用例的有效性和覆盖率，序列图的测试用例设计需要覆盖序列图的组成部分以及相关路径，相关的测试覆盖准则如下。

1. 对象覆盖准则

对象覆盖准则要求测试用例需要覆盖序列图中的所有对象，该对象可以是系统的执行者，也可以是一个类或组件。每个对象至少被访问一次，也就是至少涉及对象的一个消息传递。

以图 7-28 为例，图中有三个对象{O1，O2，O3}。根据对象覆盖准则，路径 M1-M2 或 M4-M5 都可以覆盖所有对象。但实际上，并没有覆盖全部的消息，需要其他的覆盖准则来进行测试。

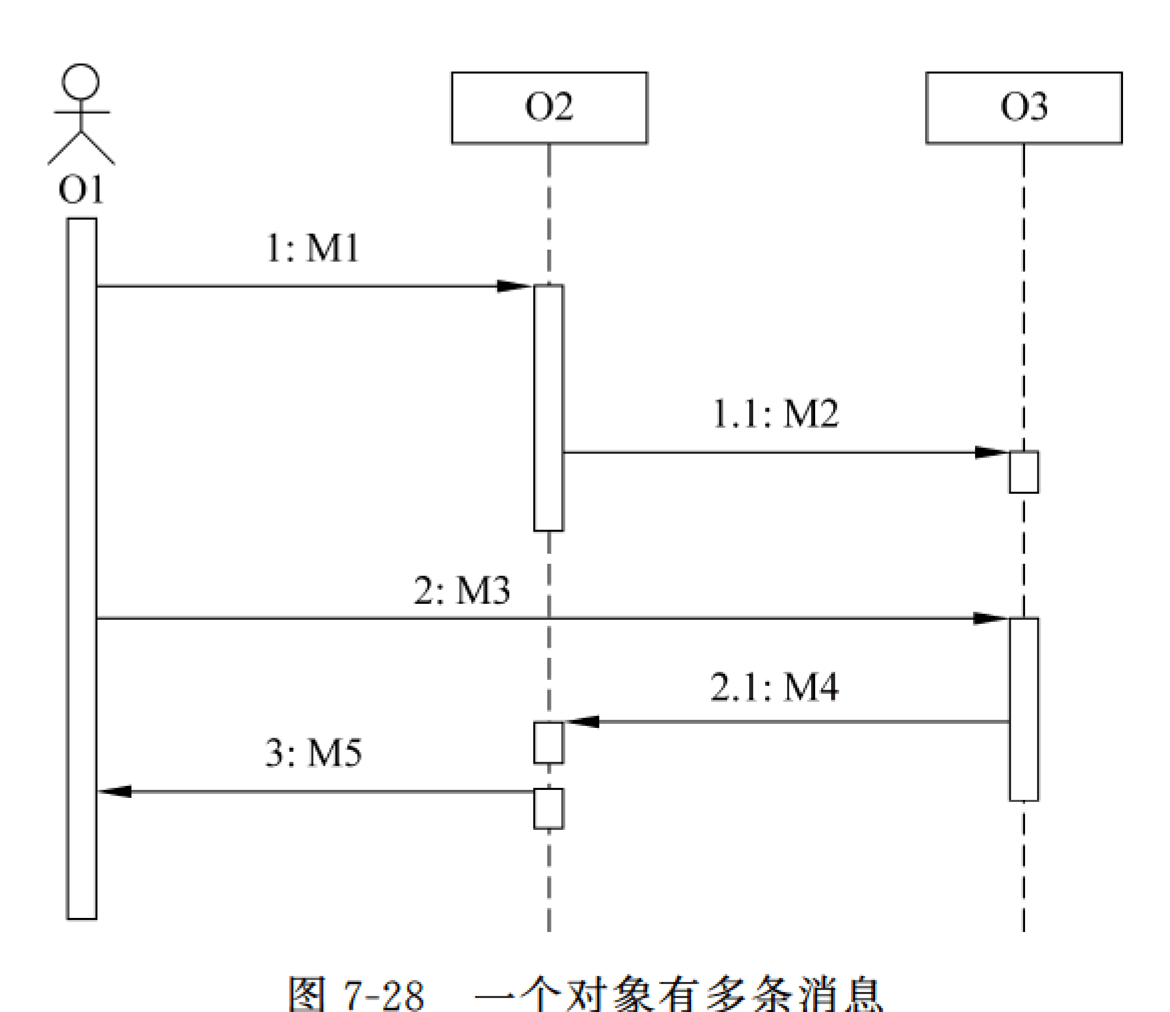


图 7-28 一个对象有多条消息

2. 消息覆盖准则

消息覆盖准则要求覆盖序列图中的所有消息，一个消息至少执行一次。消息在两个对象之间进行传递，覆盖了所有消息也就覆盖所有消息传递和接收的对象。通常情况下，序列图中没有单独存在的对象，因此消息覆盖准则可以保证对象覆盖准则。图 7-28 中有{M1，M2，M3，M4，M5}5 条消息，一条路径可以覆盖所有消息：M1-M2-M3-M4-M5。

消息覆盖准则存在着不足之处，消息覆盖不能保证路径覆盖。以图 7-29 为例，该图中有两个备选组合片段{F1，F2}，并且这两个组合片段之间是相互独立的。两条路径可

以达到消息覆盖，如下所示。

路径 1：M1-M4-M5

路径 2：M2-M3-M6

可以发现，这两条路径覆盖了所有的消息，但是没有覆盖所有的路径。实际图中还有另外两条路径没有覆盖：M1-M6 以及 M2-M3-M4-M5。

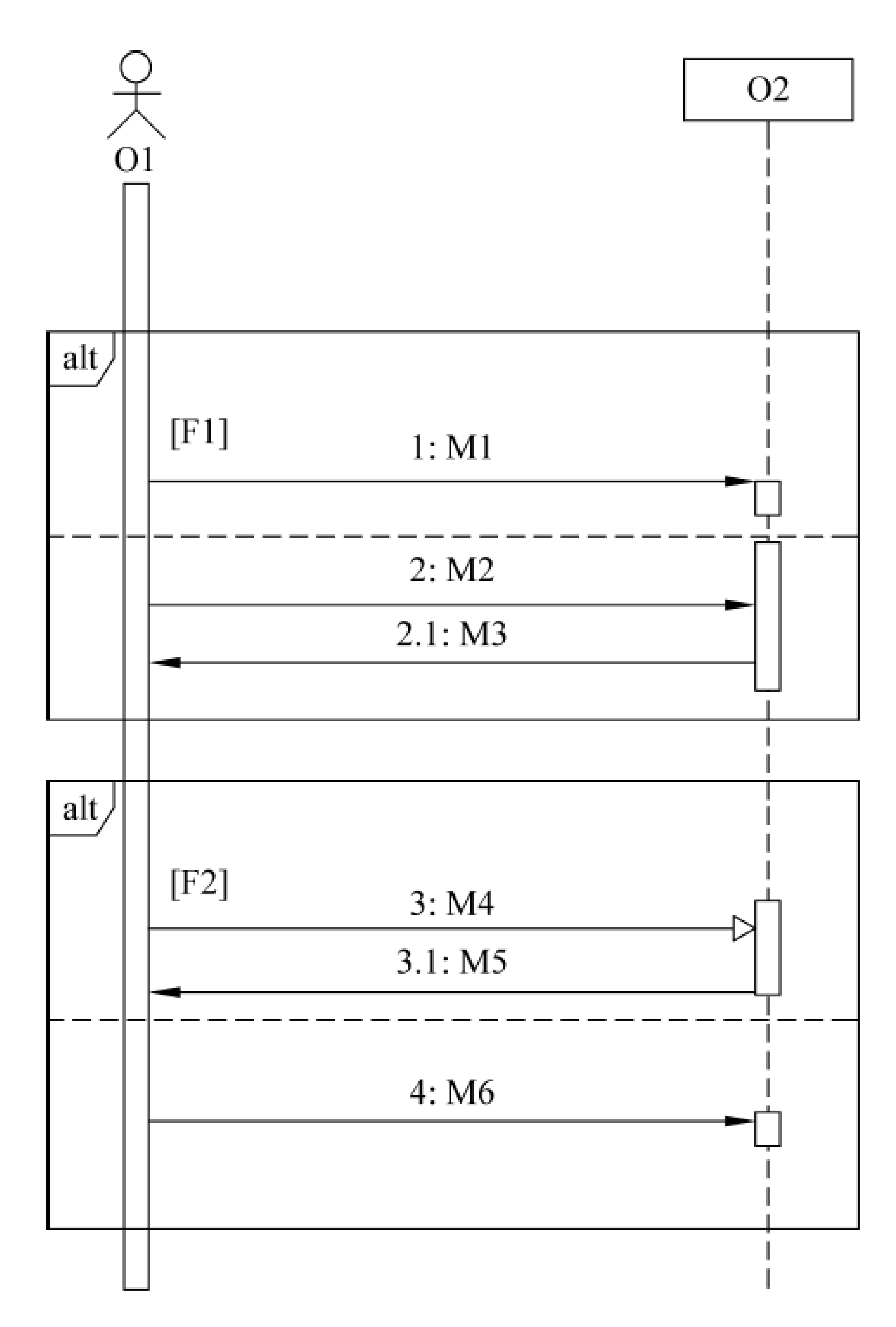


图 7-29　两个备选组合片段

3. 路径覆盖准则

路径覆盖准则要求覆盖序列图中的所有路径。序列图中的路径表示消息执行的顺序，不同测试用例描述了不同的路径。对应系统功能的每条路径都至少执行一次。路径覆盖准则可以较好地保证测试的完整性，可以弥补消息覆盖准则的缺陷。

在序列图中，消息的执行是有一定的时间顺序的，在实施路径覆盖时，必须要保证消息的正确顺序。可以定义两个消息 M_i 和 M_j，如果 M_i 的执行顺序先于 M_j，则记为 $M_i < M_j$。判断消息的顺序关系有以下三个性质。

性质一：消息 M_i 的生命线位于消息 M_j 的生命线上方，则 $M_i < M_j$。

性质二：组合片段 F_i 的生命线位于组合片段 F_j 的生命线上方，则 $F_i < F_j$。

性质三：当存在组合片段时，组合片段中不同区域的消息没有先后顺序之分，在同一个区域的消息有顺序关系。

性质四：两个组合的顺序将决定一个组合内部消息和另一个组合内部消息的顺序。

例如,图7-29中有两个备选组合片段。其中,组合片段F1中M1和M2,M1和M3没有前后顺序,因为这些消息在不同的区域。但是M2和M3在同一个区域中,并且M2的生命线在M3的生命线的上方,因此M2在M3前面执行,即M2<M3。同理,组合片段F2内部的顺序关系是M4<M5。另外,组合片段F1在组合片段F2上方,因此F1<F2。如果F1中选取消息M1,F2中选取消息M6,因为F1<F2,则M1<M6。确定了序列图中的消息顺序,可以得到以下4条路径(所有路径必须满足消息的顺序关系)。

路径1:M1-M4-M5

路径2:M2-M3-M6

路径3:M1-M6

路径4:M2-M3-M4-M5

4. 分支区域覆盖准则

分支区域覆盖准则要求覆盖序列图中所有判断后选择的分支区域。序列图中作为判断的是两个组合片段,分别是备选组合片段和选项组合片段。备选组合片段中有可选的控制流,将整个组合片段分成多个分支区域,可以进行不同的选择,但不能同时选择。选项组合片段只有一个分支区域,可以选择执行或者不执行。

对于备选组合片段,一个组合片段存在多个分支区域,在实际测试过程中,必须覆盖每个分支区域至少一次。备选组合片段中可以嵌套其他的备选组合片段。以图7-30为例,备选组合片段F3嵌套了备选组合片段F1和F2。

对于图7-30,覆盖所有判断,其实就是要覆盖每个备选组合片段的分支区域。图中F1、F2、F3都有两个分支区域,具体如下。

F1的分支区域1:M2-M3

F1的分支区域2:M4-M5

F2的分支区域1:M6-M7

F2的分支区域2:M8

F3的分支区域1:执行F1中的一个分支区域

F3的分支区域2:执行F1中的一个分支区域

分支区域覆盖准则要求序列图中的每个分支区域都要覆盖,在这个序列图中只要覆盖了F1和F2,就会覆盖F3。因为F3的执行依赖其他两个组合片段的执行。结合路径覆盖准则,用4条路径可以覆盖所有的分支区域,如下。

路径1:M1-M2-M3

路径2:M1-M4-M5

路径3:M1-M6-M7

路径4:M1-M8

选项组合片段的条件可能满足也可能不满足,因此分支区域可以执行也可以不执行,执行次数为0次和1次。当一个图中有多个备选组合片段和选项组合片段时,且互相之间是独立的,没有嵌套关系,为了覆盖每个分支区域,结合路径覆盖准则,要以组合的方式来查找路径。如果在一个序列图中有一个备选组合片段,区域数为m,有一个选项组合片

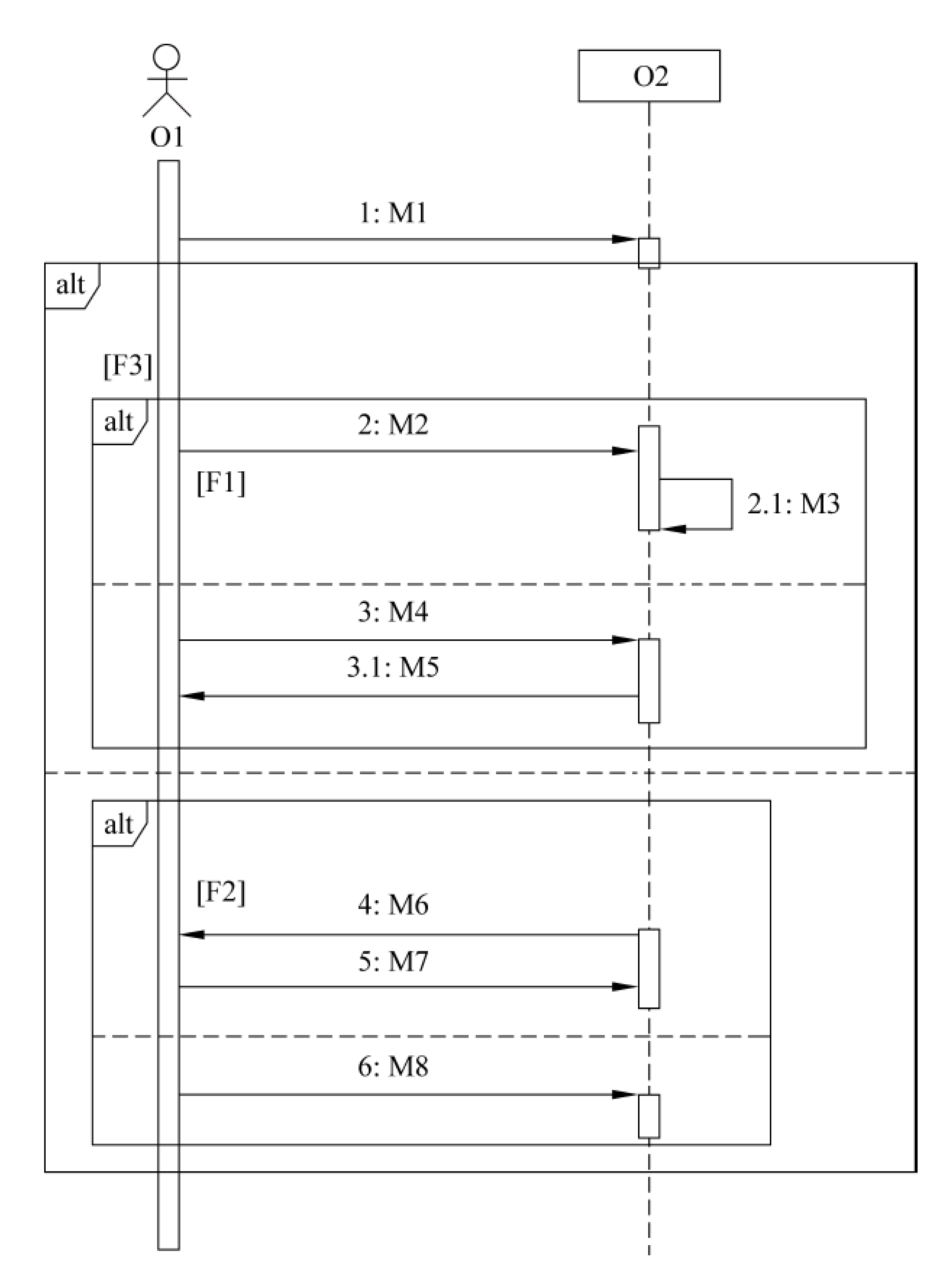


图 7-30　备选组合片段嵌套

段，有两种选择，组合得到的总路径数 $m\times2$ 条。图 7-31 中有两个备选组合片段和一个选项组合片段，则路径数为 $2\times2\times2=8$。

图 7-31 的备选组合片段 F1 和 F2 分别都有两个区域。备选组合片段 F1 有两个分支：M1-M2 和 M3。备选片段 F2 也有两个分支：M4-M5 和 M6。选项组合片段 F3 可以执行也可以不执行，因此也有两种情况。将各个片段进行组合，可以得到以下 $2\times2\times2=8$ 条路径。

路径 1：M1-M2-M4-M5

路径 2：M1-M2-M4-M5-M7-M8-M9

路径 3：M1-M2-M6

路径 4：M1-M2-M6-M7-M8-M9

路径 5：M3-M4-M5

路径 6：M3-M4-M5-M7-M8-M9

路径 7：M3-M6

路径 8：M3-M6-M7-M8-M9

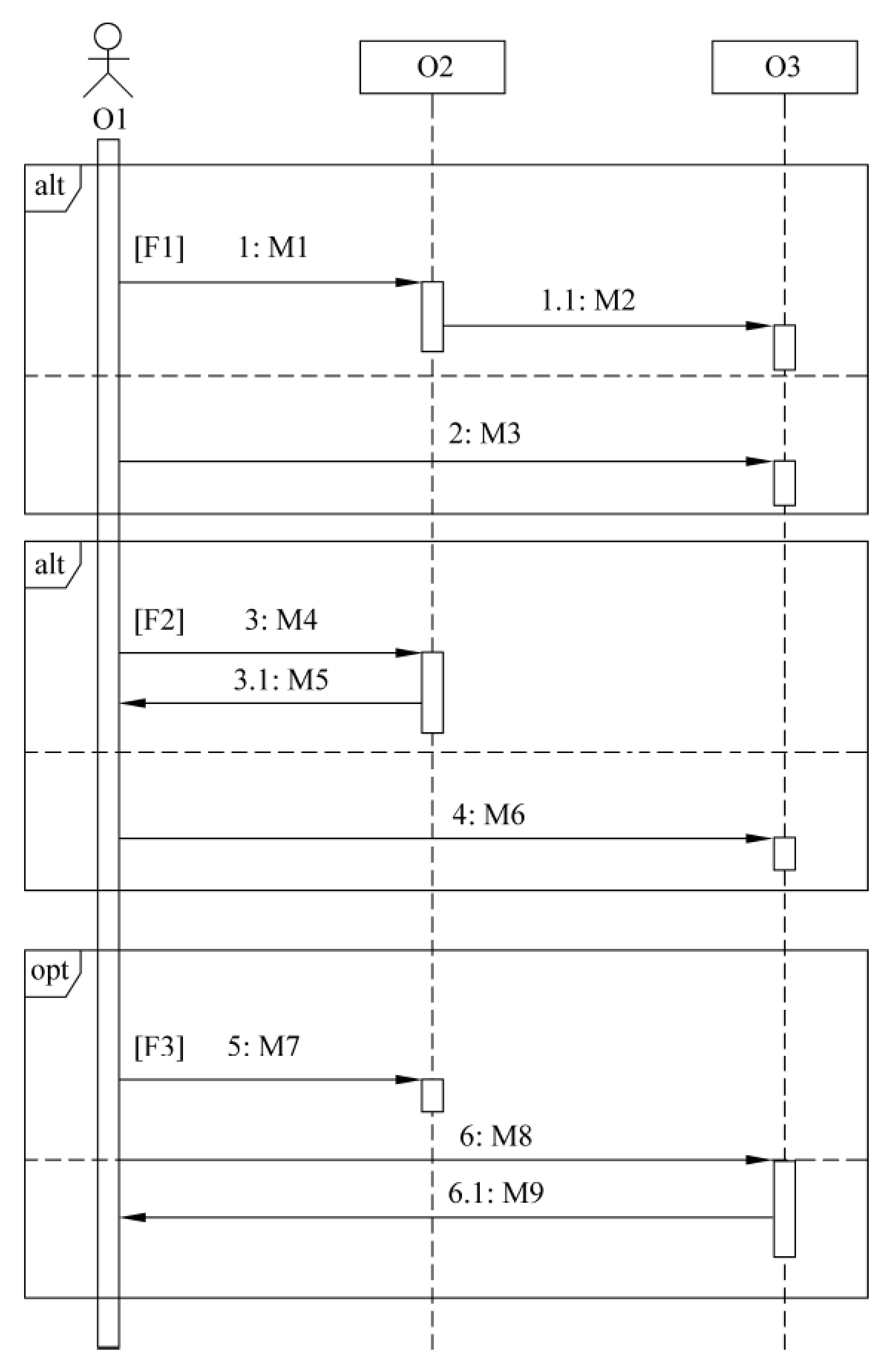


图 7-31 多个独立组合片段

5. 循环覆盖准则

序列图中的循环组合片段将执行多次区域内的消息，覆盖循环的测试用例需要覆盖多次消息执行。如果已经标明执行的次数，则按标明的次数来执行。如果没有标明执行的次数，则可以执行 0 次、1 次和 *n* 次该片段。*n* 的值根据实际情况来确定，不能过大，否则会影响测试效率。

循环组合片段可能会嵌套另外一个循环组合片段，情况会比较复杂。以图 7-32 为例，图中有两个循环组合片段 F1 和 F2，其中 F1 嵌套 F2，都没有标明执行的次数，选择执行 0 次、1 次和两次。

首先，外部循环 F1 可以执行 0 次、1 次和两次。当 F1 执行 0 次时，F2 默认执行 0 次。当 F1 执行一次时，F2 可以执行 0 次、1 次和两次，总计三种情况。当 F1 执行两次时，需要将这两次分开分析，每次 F2 都可以执行 0 次、1 次和两次，总计 3×3=9 种情况。可以得到 1+3+9=13 路径，如下。

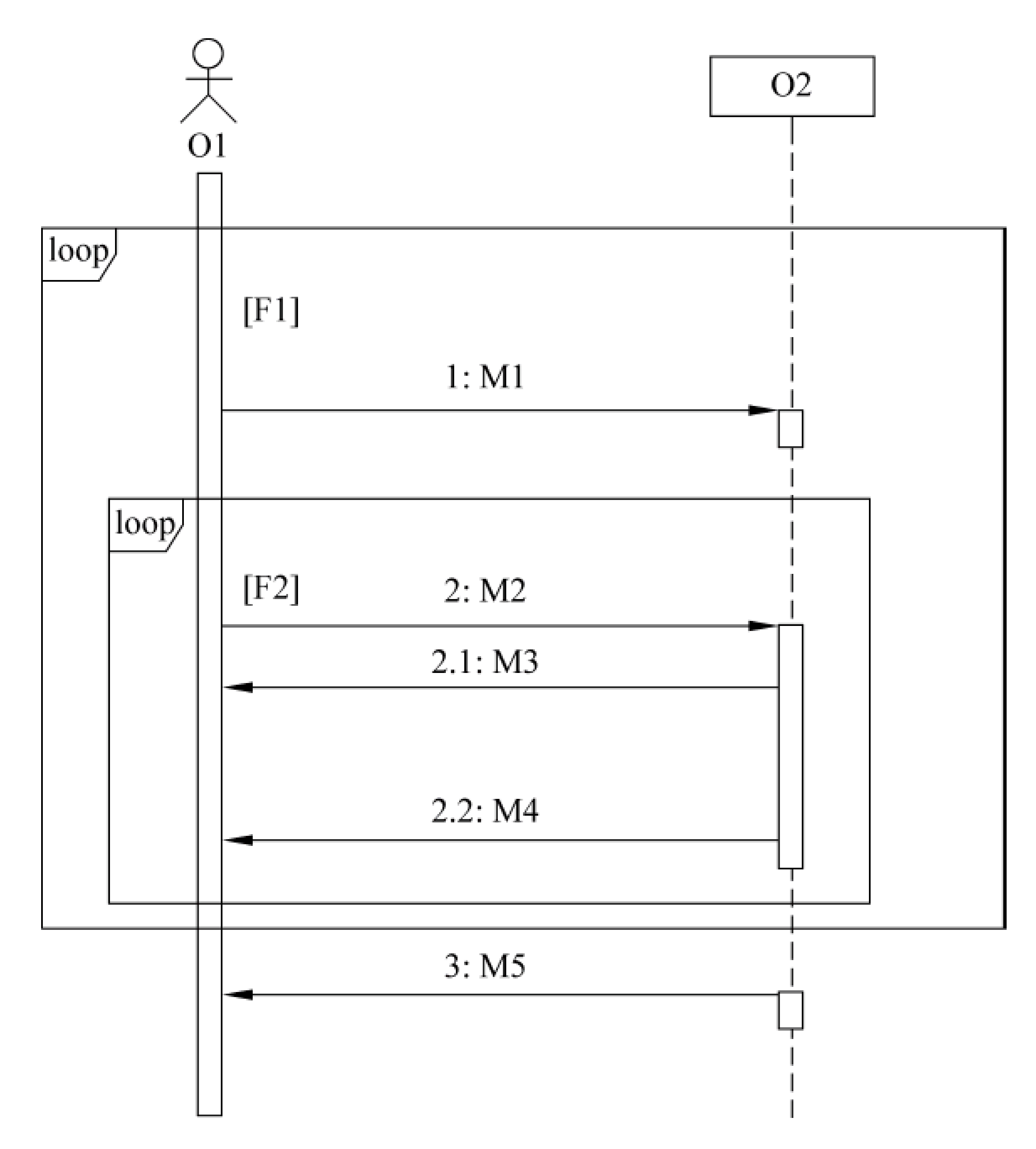


图 7-32　循环组合片段嵌套

路径 1：M5

路径 2：M1-M5

路径 3：M1-M2-M3-M4-M5

路径 4：M1-M2-M3-M4-M2-M3-M4-M5

路径 5：M1-M1-M5

路径 6：M1-M1-M2-M3-M4-M5

路径 7：M1-M1-M2-M3-M4-M2-M3-M4-M5

路径 8：M1-M2-M3-M4-M1-M5

路径 9：M1-M2-M3-M4-M1-M2-M3-M4-M5

路径 10：M1-M2-M3-M4-M1-M2-M3-M4-M2-M3-M4-M5

路径 11：M1-M2-M3-M4-M2-M3-M4-M1-M5

路径 12：M1-M2-M3-M4-M2-M3-M4-M1-M2-M3-M4-M5

路径 13：M1-M2-M3-M4-M2-M3-M4-M1-M2-M3-M4-M2-M3-M4-M5

多重嵌套的情况更加复杂，需要仔细分析。一层一层地剥离循环，先将外部循环分析清楚，再依次分析内部循环。具体路径查找方法可参照第 3 章中的路径覆盖。

7.5.3　序列图的测试用例设计

根据 UML 序列图设计具体的测试用例，设计步骤如下。

(1) 根据软件规格说明书，获得序列图。

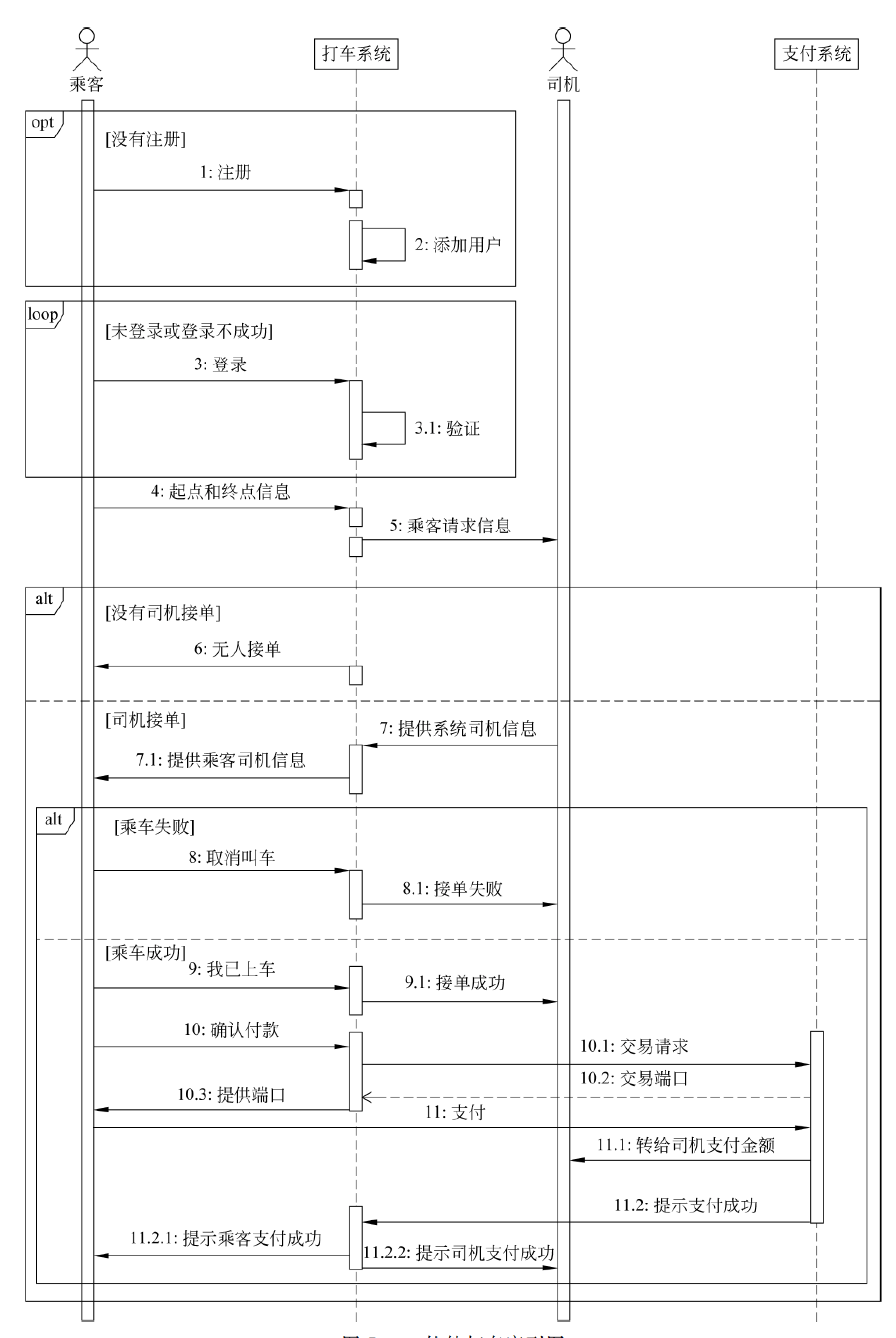


图 7-33 软件打车序列图

M18：支付

M19：转给司机支付金额

M20：提示支付成功

M21：提示乘客支付成功

M22：提示司机支付成功

交互框有 4 个，如下。

F1：没有注册

F2：未登录或登录不成功

F3：是否有司机接单

F4：乘车是否成功

步骤三：根据序列图的覆盖准则设计测试用例，主要采用路径覆盖准则、判断覆盖准则和循环覆盖准则，这三个准则包括对象覆盖准则和消息覆盖准则。

采用路径覆盖准则，需要明确消息的顺序关系。由图可知，整体的顺序关系是：F1＞F2＞M5＞M6＞F3。在 F1 和 F2 内部的顺序需要按照时间先后关系排序。F3 内部嵌套了一个交互框 F4，具体的路径查找需要依据判断覆盖准则。在查找路径的过程中，必须要遵循消息的顺序关系。

根据判断覆盖准则，F1 是一个选项组合片段，有两种选择：执行或者不执行。F3、F4 是两个备选组合片段，其中 F3 中嵌套了 F4。F4 有两个区域，F3 也有两个区域，因此 F3 执行的选择有 1＋2＝3 种。

序列图中存在循环，需要符合循环覆盖准则，本例选择循环执行次数为 0 次、1 次和两次，总共三种情况。由于 F1、F2 和 F3 是相互独立的，因此实际路径数为 2×3×3＝18，如表 7-17 所示。

表 7-17 软件打车的测试用例

用例编号	测试路径	条件
1	起点和终点信息→乘客请求信息→无人接单	[已注册] [已登录] [无人接单]
2	起点和终点信息→乘客请求信息→提供系统司机信息→提供乘客司机信息→取消叫车→接单失败	[已注册] [已登录] [司机接单] [乘车失败]
3	起点和终点信息→乘客请求信息→提供系统司机信息→提供乘客司机信息→我已上车→接单成功→确认付款→交易请求→交易端口→提供端口→支付→转给司机支付金额→提示支付成功→提示乘客支付成功→提示司机支付成功	[已注册] [已登录] [司机接单] [乘车成功]
4	登录→验证→起点和终点信息→乘客请求信息→无人接单	[已注册] [未登录] [已登录] [无人接单]

续表

用例编号	测 试 路 径	条　　件
5	登录→验证→起点和终点信息→乘客请求信息→提供系统司机信息→提供乘客司机信息→取消叫车→接单失败	[已注册] [未登录] [已登录] [司机接单] [乘车失败]
6	登录→验证→起点和终点信息→乘客请求信息→提供系统司机信息→提供乘客司机信息→我已上车→接单成功→确认付款→交易请求→交易端口→提供端口→支付→转给司机支付金额→提示支付成功→提示乘客支付成功→提示司机支付成功	[已注册] [未登录] [已登录] [司机接单] [乘车成功]
7	登录→验证→登录→验证→起点和终点信息→乘客请求信息→无人接单	[已注册] [未登录] [登录不成功] [已登录] [无人接单]
8	登录→验证→登录→验证→起点和终点信息→乘客请求信息→提供系统司机信息→提供乘客司机信息→取消叫车→接单失败	[已注册] [未登录] [登录不成功] [已登录] [司机接单] [乘车失败]
9	登录→验证→登录→验证→起点和终点信息→乘客请求信息→提供系统司机信息→提供乘客司机信息→我已上车→接单成功→确认付款→交易请求→交易端口→提供端口→支付→转给司机支付金额→提示支付成功→提示乘客支付成功→提示司机支付成功	[已注册] [未登录] [登录不成功] [已登录] [司机接单] [乘车成功]
10	注册→添加用户→起点和终点信息→乘客请求信息→无人接单	[未注册] [已登录] [无人接单]
11	注册→添加用户→起点和终点信息→乘客请求信息→提供系统司机信息→提供乘客司机信息→取消叫车→接单失败	[已注册] [已登录] [司机接单] [乘车失败]
12	注册→添加用户→起点和终点信息→乘客请求信息→提供系统司机信息→提供乘客司机信息→我已上车→接单成功→确认付款→交易请求→交易端口→提供端口→支付→转给司机支付金额→提示支付成功→提示乘客支付成功→提示司机支付成功	[未注册] [已登录] [司机接单] [乘车成功]

表 7-18 测试用例 9

步骤	输 入	预 期 输 出
1	乘客输入用户名 user,密码 34	登录不成功
2	乘客输入用户名 user,密码 12	登录成功
3	乘客输入起点(南京路步行街)和终点(东方明珠电视塔)	司机收到请求信息
4	司机单击“接单”	乘客收到司机信息
5	乘客单击“我已上车”	显示接单成功
6	乘客输入 21 元,确认付款	提示支付成功,乘客账户少 21 元,司机账户多 21 元

7.6 基于状态图的软件测试方法

7.6.1 状态图的概念

状态图,又称为状态机图,用于描述一个类/对象、一个用例甚至一个系统的状态和事件引起的状态转移。状态通常描述对象生命周期内某一时间段的状况,同一对象或者不同对象会有不同的状态。随着时间的变迁,会触发某些事件,从而导致状态的转移。活动图和状态图都是 UML 动态建模的图形工具,但状态图不同于活动图,活动图强调的是发生的动作和动作产生的结果,而状态图更侧重于状态的变化。

状态图展示了不同情况下对象不同的状态,并且把状态的变化表现出来。状态的变化是由一些事件的触发引起的,一般也可以认为这些事件是状态转移的条件。

状态图由 5 部分组成,分别是状态、转移、事件、活动和动作。这 5 部分的关系是:转移是状态之间的转化;事件是触发状态转移的条件;活动是状态内部需要执行的功能;动作是活动的组成部分。

状态描述的是一个对象某一时间段内的状况,可以显示该对象正在执行的动作或者正在等待的某些事件。例如,电梯在下降是电梯在某一时间段内的状态。一个状态可以包含其他的子状态,这种状态称为组合状态或者复合状态。同时,状态可能包含一系列的动作,例如进入状态时触发的动作、激活状态下执行的动作和离开状态时触发的动作。这些动作可以描述状态内部需要执行的功能。

根据发生时间划分状态可分为初始状态、一般状态和终止状态。

(1) 初始状态,是指状态图的起点,是对象开始的状态。

(2) 一般状态,是排除初始状态和终止状态的对象状态。需要标明状态名称,必要时要说明状态内部的动作和活动。

(3) 终止状态,是状态的终点,说明所有的状态转化都结束了,因此终止状态不会指向任何的状态。

4. 时间事件

时间事件是由时间引起的事件。开始的时候，可能没有触发这个事件，但是经过一段时间或者到了某个时间点，可能会触发这个事件。例如，用户在银行取消办理短信提醒业务(用户账户内金额发生变化时会短信提醒)，该事件不会马上触发，会到下个月开始的时候才会触发，这就是一个时间事件。

活动和动作是发生在状态内部的，动作是活动的组成部分，即活动是由一系列的动作组成。而区别在于动作具有原子性，是不可分割不可中断的，而活动是可中断的。在状态内部，有三个主要的动作和活动：进入状态时触发的动作、状态处于激活状态时的动作或活动和退出状态时的动作。这三个动作和活动贯穿了整个状态，同时所有动作和活动都必须在状态结束前结束，包括退出状态时的动作。

对序列图中的各个元素进行形式化定义，以方便识别与操作，具体定义如下。

(1) S 表示序列图中所有状态的非空有限集合，表示如下：

$$S=\{S_1,S_2,\cdots,S_i,\cdots,S_n\}$$

其中，S_i 表示一个消息，标号为 i。初始状态用 initial 表示，终止状态用 final 表示。

(2) T 表示序列图中的所有转移的非空有限集合，表示如下：

$$T=\{T_1,T_2,\cdots,T_i,\cdots,T_n\}$$

其中，T_i 表示一个消息，标号为 i。

状态图所涉及的图标，如表 7-19 所示。

表 7-19　状态图的图标表示

名　　称	解　　释	图示
状态	一个对象某一时间段内的状况	
初始状态	状态图的起点	
终止状态	状态图的终点	
浅层历史状态	当前状态的前一个历史状态	
深层历史状态	当前状态之前的历史状态	
判断选择	不同判断结果指向不同的状态	
分叉	一个状态转移到多个状态	
结合	多个状态转移到一个状态	
连接	连接多个转移	
终止	状态机的执行终止	
转移	从一个状态转换到另一个状态	
注释	对元素进行解释	

要覆盖图中的所有转移,可以设计以下三个转移路径。

路径 1:initial-S1-S4-S5-final

路径 2:initial-S2-S4-S6-final

路径 3:initial-S3-S4-S6-final

这三个测试覆盖了所有的转移,但是可以看到状态 S1 转移到 S4,再转移到 T8 没有被测试。S2 转移到 S4 再转移到 S5,以及 S3 转移到 S4 再转移到 S6,这两条路径也没有被测试。

3. 转移对覆盖准则

转移对覆盖准则要求测试用例覆盖状态图中所有的转移对。所谓转移对,是指两个相邻的转移。如果一个转移有多个相邻的转移,则必须要同时测试该转移和所有相邻转移,不能因为一个转移已经测试过,而不进行第二次测试。

对于图 7-35,根据转移对覆盖准则,可以设计以下 6 个转移路径。

路径 1:initial-S1-S4-S5-final

路径 2:initial-S1-S4-S6-final

路径 3:initial-S2-S4-S5-final

路径 4:initial-S2-S4-S6-final

路径 5:initial-S3-S4-S5-final

路径 6:initial-S3-S4-S6-final

可以看出,这 6 个转移序列覆盖了所有的转移对:{T1,T4}、{T2,T5}、{T3,T6}、{T4,T7}、{T4,T8}、{T5,T7}、{T5,T8}、{T6,T7}、{T6,T8}、{T7,T9}和{T8,T10}。

转移对覆盖准则可以保证状态转移分支的覆盖。当一个状态通过判断可以转移到多个不同的状态时,为了覆盖转移对,必须要覆盖每个分支。

以图 7-36 为例,图中有一个判断,状态 S1 可以通过判断转移到 S2 和 S4。为了保证转移对状态覆盖准则,转移对{T2,T3}和{T2,T4}都必须被覆盖。因此,两个分支都会被覆盖,保证了分支覆盖。

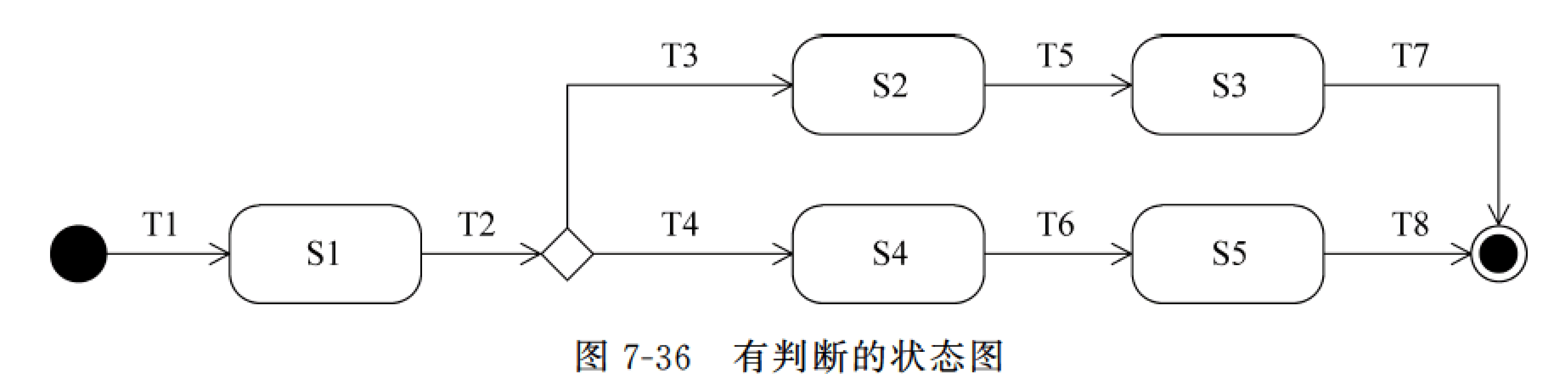


图 7-36　有判断的状态图

4. 路径覆盖准则

路径覆盖准则要求测试用例覆盖从开始状态到终止状态的所有路径,每条路径都必须检测一次。这里的路径指的是有效路径,不能只关注状态图中直观的路径,而忽略了到达下一个状态的条件。虽然状态图中一个状态可以转移到另一个状态,但可能需要经过其他的状态才能走这一条路径,而不能直接转移。

(1) 根据软件规格说明书,画图得到状态图。

(2) 分析状态图的结构,识别每个状态和转移边。

(3) 根据状态图的覆盖准则,找到多条路径,每条路径对应一个测试用例。

(4) 根据路径以及需要满足的条件,设计测试用例。

下面以团购系统为例,说明状态图的测试用例设计过程。

步骤一:根据软件规格说明书,得到状态图。下面是团购系统的需求说明。

(1) 进入团购时,系统会自动定位所在地。如果无法定位,可手动输入。确认位置后,仍然可以修改。

(2) 团购的种类限定为三种:KTV、美食和电影。

(3) 选择KTV团购或者美食团购,需要选择相应的团购券,等待确认订单。在没有确认订单之前可以选择重新下单。

(4) 选择电影团购,需要选择电影和电影院,并且要选座,等待确认订单。在没有确认订单之前可以重新选座或者重新选择电影和电影院。

(5) 确认订单支付成功后,系统会提示下单成功,并且返回订单号。

根据上述描述,可得状态图如图7-39所示。

步骤二:得到状态图后,需要分析状态图中的组成元素,主要分析状态和转移。图7-39中有15个状态和22种转移。

状态有15个,如下。

S1:初始状态

S2:未知地点

S3:已知地点

S4:KTV团购界面

S5:等待抢购KTV团购券

S6:美食团购界面

S7:等待抢购美食团购券

S8:电影团购界面

S9:显示价格和时间

S10:显示座位情况

S11:等待提交

S12:等待确认订单

S13:等待支付

S14:提示成功下单

S15:终止状态

转移有22种,如下。

T1:无法自动定位

T2:自动定位

T3:手动输入

T4:修改地点

T9：选择美食

T10：选择美食团购券

T11：选择抢购美食团购券

T12：选择重新下单(美食)

T13：选择电影

T14：选择电影和影院

T15：选择选座购票

T16：选定座位

T17：提交座位信息

T18：重新选座

T19：重选电影和电影院

T20：确认订单

T21：支付成功

T22：结束交易

步骤三：根据状态图的覆盖准则设计测试用例，这里主要采用转移对覆盖准则、路径覆盖准则和循环覆盖准则。这三个准则保证了状态覆盖准则和转移覆盖准则。

状态图的整体分布可分为三个部分：第一部分是检测位置，第二部分是下订单，第三部分是支付订单。状态(已知地点)前面是第一部分，状态(已知地点)到状态(等待确认订单)是第二部分，剩余的状态是第三部分。根据转移对覆盖准则，转移到状态 S3 和从状态 S3 转移到其他状态涉及的转移对都要被覆盖。也就意味着位置确定后的三种团购都需要测试。同理，确定订单后，三种团购方式都要进行统一的支付。

根据路径覆盖准则，从开始状态出发有两条路径可走，分别是无法自动定位和自动定位。第一部分的检测位置中存在一个循环，根据循环覆盖准则，需要覆盖循环 0 次、1 次和 n 次，这里直接取 0 次和 1 次两种情况。因此，第一部分总共有 $2\times2=4$ 种情况，分别是 S1-S2-S3、S1-S2-S3-S3、S1-S3 和 S1-S3-S3。

第二部分中有三个分支，需要一一进行分析。选择 KTV 和选择美食到等待确认订单都只有一个循环。根据循环覆盖准则，选择覆盖次数 0 次和 1 次，分别有两种情况。选择电影院到等待确认订单存在两个循环，并且大的循环嵌套了一个小的循环。从状态 S3 到状态 S12，共有 5 种情况，分别是两个循环都不覆盖、覆盖两个循环其中的一个循环一次和两个循环都覆盖一次(顺序有先后)。第二部分总共有 $2+2+5=9$ 条路径。

第三部分只有一条路径。因此整个状态图的路径数为 $4\times9\times1=36$ 条，如表 7-20 所示。

表 7-20　团购的测试用例

用例编号	测试路径	条　　件
1	初始状态→未知地点→已知地点→KTV 团购界面→等待抢购 KTV 团购券→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [选择 KTV]

续表

用例编号	测试路径	条　件
2	初始状态→未知地点→已知地点→KTV团购界面→等待抢购KTV团购券→等待确认订单→等待抢购KTV团购券→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [选择KTV] [选择重新下单(KTV)]
3	初始状态→未知地点→已知地点→美食团购界面→等待抢购美食团购券→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [选择美食]
4	初始状态→未知地点→已知地点→美食团购界面→等待抢购美食团购券→等待确认订单→等待抢购美食团购券→等待确认订单→等待支付→提示成功下单→终止状	[无法自动定位] [选择美食] [选择重新下单美食]
5	初始状态→未知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [选择电影]
6	初始状态→未知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [选择电影] [重新选座]
7	初始状态→未知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [选择电影] [重选电影和电影院]
8	初始状态→未知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→显示座位情况→等待提交→等待确认订单→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [选择电影] [重新选座] [重选电影和电影院]
9	初始状态→未知地点→已知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择电影] [重选电影和电影院] [重新选座]
10	初始状态→未知地点→已知地点→已知地点→KTV团购界面→等待抢购KTV团购券→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择KTV]
11	初始状态→未知地点→已知地点→已知地点→KTV团购界面→等待抢购KTV团购券→等待确认订单→等待抢购KTV团购券→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择KTV] [选择重新下单(KTV)]
12	初始状态→未知地点→已知地点→已知地点→美食团购界面→等待抢购美食团购券→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择美食]

续表

用例编号	测试路径	条　件
13	初始状态→未知地点→已知地点→已知地点→美食团购界面→等待抢购美食团购券→等待确认订单→等待抢购美食团购券→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择美食] [选择重新下单(美食)]
14	初始状态→未知地点→已知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择电影]
15	初始状态→未知地点→已知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择电影] [重新选座]
16	初始状态→未知地点→已知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择电影] [重选电影和电影院]
17	初始状态→未知地点→已知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→显示座位情况→等待提交→等待确认订单→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择电影] [重新选座] [重选电影和电影院]
18	初始状态→未知地点→已知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[无法自动定位] [修改地点] [选择电影] [重选电影和电影院] [重新选座]
19	初始状态→已知地点→KTV团购界面→等待抢购KTV团购券→等待确认订单→等待支付→提示成功下单→终止状态	[自动定位] [选择KTV]
20	初始状态→已知地点→KTV团购界面→等待抢购KTV团购券→等待确认订单→等待抢购KTV团购券→等待确认订单→等待支付→提示成功下单→终止状态	[自动定位] [选择KTV] [选择重新下单(KTV)]
21	初始状态→已知地点→美食团购界面→等待抢购美食团购券→等待确认订单→等待支付→提示成功下单→终止状态	[自动定位] [选择美食]
22	初始状态→已知地点→美食团购界面→等待抢购美食团购券→等待确认订单→等待抢购美食团购券→等待确认订单→等待支付→提示成功下单→终止状态	[自动定位] [选择美食] [选择重新下单(美食)]
23	初始状态→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[自动定位] [选择电影]

续表

用例编号	测试路径	条件
34	初始状态→已知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[自动定位] [修改地点] [选择电影] [重选电影和电影院]
35	初始状态→已知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→显示座位情况→等待提交→等待确认订单→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[自动定位] [修改地点] [选择电影] [重新选座] [重选电影和电影院]
36	初始状态→已知地点→已知地点→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→电影团购界面→显示价格和时间→显示座位情况→等待提交→等待确认订单→显示座位情况→等待提交→等待确认订单→等待支付→提示成功下单→终止状态	[自动定位] [修改地点] [选择电影] [重选电影和电影院] [重新选座]

步骤四：根据路径以及需要满足的条件，设计测试用例。

以测试用例33为例，假设现有一个用户所在地为上海，团购的地点是杭州。用户想要团购两张电影票，电影院为横店电影院，电影为超能陆战队，则设计的测试用例具体如表7-21所示。

表7-21　测试用例33

步骤	输　　入	预期输出
1	系统自动定位	显示地点为上海
2	用户修改地点为杭州	显示地点为杭州
3	用户单击“电影类别”	显示电影团购界面
4	用户选择横店电影院的超能陆战队	显示价格和时间
5	选择19点场次的选座购票	显示座位情况
6	选择两个座位(E排06,07)	显示订单，等待确认
7	选择重新选座	显示座位情况
8	选择两个座位(D排06,07)	显示订单，等待确认
9	单击“确认订单”	显示支付界面
10	输入账号密码，支付成功	提示支付成功并返回订单号

(5) $P \cup T = \varnothing$，规定了网中至少有一个元素。

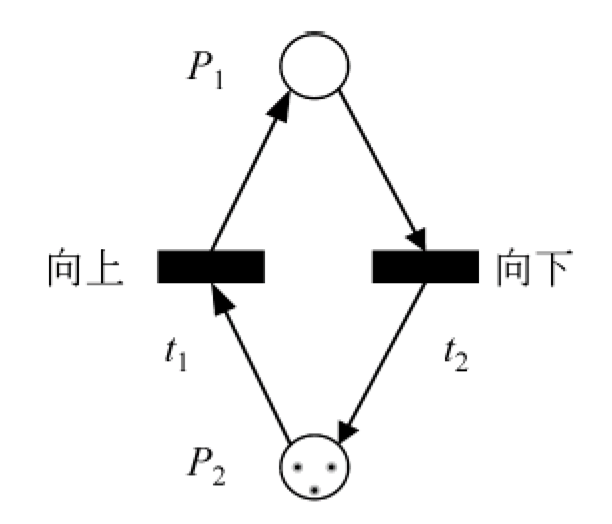
(6) $\text{dom}(F) \cup \text{cod}(F) = P \cup T$，表示在网中不能有孤立元素，每一个库所或者变迁必须通过弧和其他元素关联。

$$\text{dom}(F) = \{x \mid \exists y: (x,y) \in F\}, \quad \text{cod}(F) = \{x \mid \exists y: (y,x) \in F\}$$

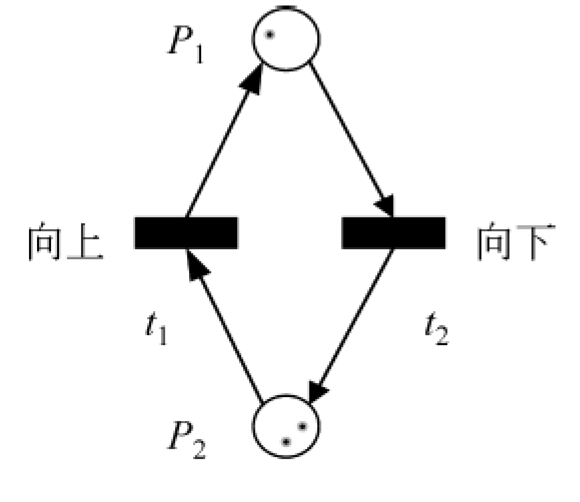
变迁的发生受到系统状态的控制，即变迁发生的前置条件必须满足；变迁发生后，某些前置条件不再满足，而某些后置条件则得到满足。

例 8-1 简单电梯的 Petri 网表示。

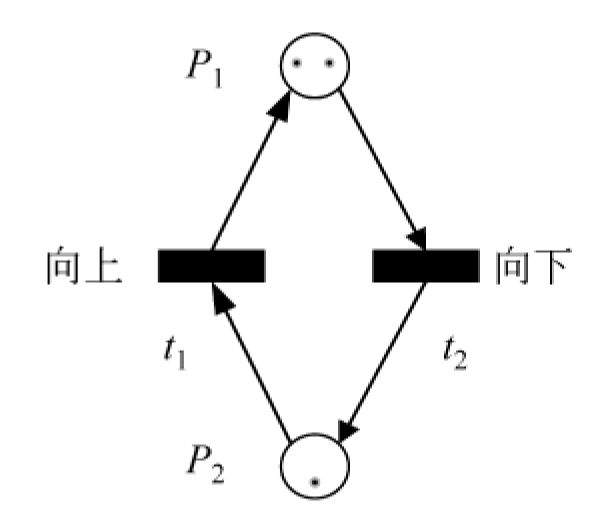
有一个三层的电梯，其电梯按钮仅包括向上和向下（无法直接选择楼层），即该电梯每次仅能够上升或者下降一层，那么该电梯的 Petri 网如图 8-1 所示。这个 Petri 网包含两个库所和两个变迁，左边变迁 t_1 表示向上按钮，右边变迁 t_2 表示向下按钮。库所 P_1 表示离地面的层数，库所 P_2 表示离楼顶的层数，楼层数分别用 Token 肯表示。其中，图 8-1(a) 表示电梯的初始状态，P_2 中间包含三个 Token，而 P_1 没有 Token，意味着向上按钮处于启用状态，而向下按钮处于禁用状态，用户只能触发向上的变迁。当用户按下向上按钮以后，电梯向上一层，P_2 的一个 Token 向 P_1 转移，P_1 和 P_2 分别包含 Token，向上按钮和向下按钮都处于启用状态，这时用户可以选择向下或者向下按钮。当用户连续按了三次向上按钮以后，P_2 里所有 Token 都转移到 P_1 中间，此时用户只能选择向下按钮，而向上按钮处于禁用状态，如图 8-1(d) 所示。



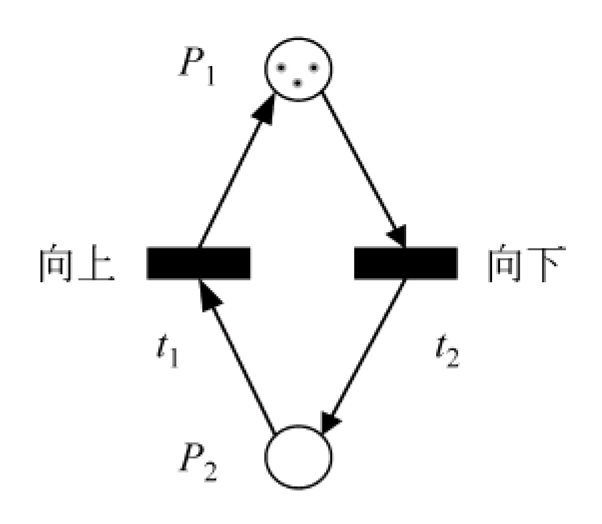
(a) 初始状态，
向上变迁处于启用状态
向下变迁处于禁用状态



(b) 向上变迁触发一次，
向上(下)变迁均处于启用状态



(c) 向上变迁触发两次，
向上(下)变迁均处于启用状态



(d) 向上变迁触发三次，
向上变迁处于禁用状态
向下变迁处于启用状态

图 8-1 一个简单电梯的 Petri 网示意图

为了叙述的方便，需要引入库所或者迁移的前集和后集的概念。若 $x \in (P \cup T)$，设

(2) P 是库所的有限集。

(3) T 是变迁的有限集。

(4) A 是弧的有限集,P、T、A 三者互不相交。

(5) N 是节点函数,其中 $N:A->(T\times P\cup T\times P)$。

(6) C 是颜色的集合,其中 $C: A->\Sigma$。

(7) G 是哨兵函数,定义为 $G: T->\mathrm{Expr}$ 且满足 $\forall t\in T:[\mathrm{Type}(G(t))=B\wedge \mathrm{Type}(\mathrm{Var}(E(a)))\subseteq\Sigma]$,其中,Type($v$)表示变量的类型,Var(Expr)表示表达式 Expr 的变量集。

(8) E 为弧表达式函数,定义为 $E: A->\mathrm{Expr}$ 且满足 $\forall a\in A:[\mathrm{Type}(E(a))=C(p(a))_{ms}\wedge \mathrm{Type}(\mathrm{Var}(E(a)))\subseteq\Sigma]$。其中,$p$ 为 $N(a)$中的位置 MS;$C(P)$ 表示位于 $C(p)$之上的所有多重集的集合。

(9) I 是初始化函数,定义为 $I: P->\mathrm{Expr}$ 且满足 $\forall a\in P:[\mathrm{Type}(I(p))=C(p)_{\mathrm{MS}}]$。

CPN Tools 工具是由丹麦 Arthus 大学开发的 CPN 建模仿真和性能分析工具。CPN Tools 提供图形化的开发环境和信息反馈,支持功能强大的元语言,它在建模过程中可以随时对模型进行语法检查,同时能够自动生成部分代码段,建立好模型后,可以利用仿真工具进行仿真并收集仿真数据得到仿真报告,同时还可以自动地生成并分析完全的或部分的状态空间,利用得到的状态空间报告和状态空间图分析模型的可达性、有界性、活性和公平性等。

哲学家就餐问题的 Petri 网描述。

哲学家问题可以用来描述死锁和资源耗尽。假设有 5 位哲学家围坐在一张圆形餐桌旁,做以下两件事情之一:吃饭或者思考。吃东西的时候,他们就停止思考,思考的时候也停止吃东西。餐桌中间有一大碗意大利面,每两个哲学家之间有一只餐叉。因为用一只餐叉很难吃到意大利面,所以假设哲学家必须用两只餐叉吃东西。他们只能使用自己左右手边的那两只餐叉。

图 8-2 是用 Petri 网表示的哲学家就餐问题,不同的筷子需要用不同的库所以及对应的 Token 表示,而每一个哲学家的思考状态和就餐状态也需要用不同的库所表示。在思考状态、就餐状态之间的转换需要两个变迁实现,在图 8-2 中右上角表示其中一个哲学家的情况,5 个哲学家共享相邻的筷子(库所)。

若用 CPN 表示哲学家就餐问题,先定义相关的函数和颜色集:

```
val n=5;
color PH=index ph with 1..n;
color CS=index cs with 1..n;
var p: PH;
fun Chopsticks(ph(i))=1`cs(i)++1`cs(if i=n then 1 else i+1);
```

这里 n 是常量 5,其中,PH 和 CS 分别表示哲学家和筷子类型,各有 5 个筷子和哲学家。

图 8-3(a) 给出了用 CPN 表示的哲学家就餐问题,这个 CPN 图显然比 Petri 网表示

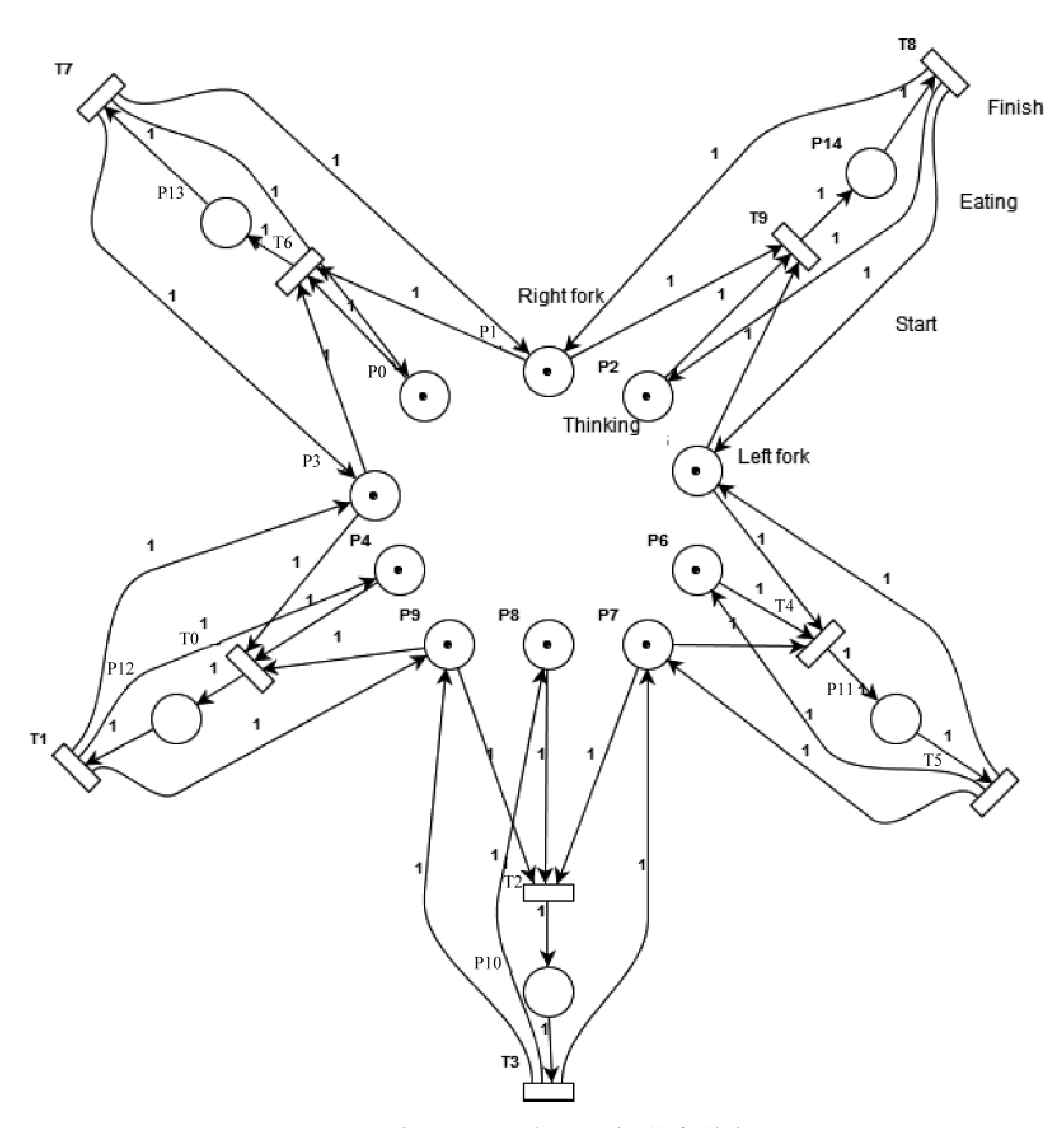
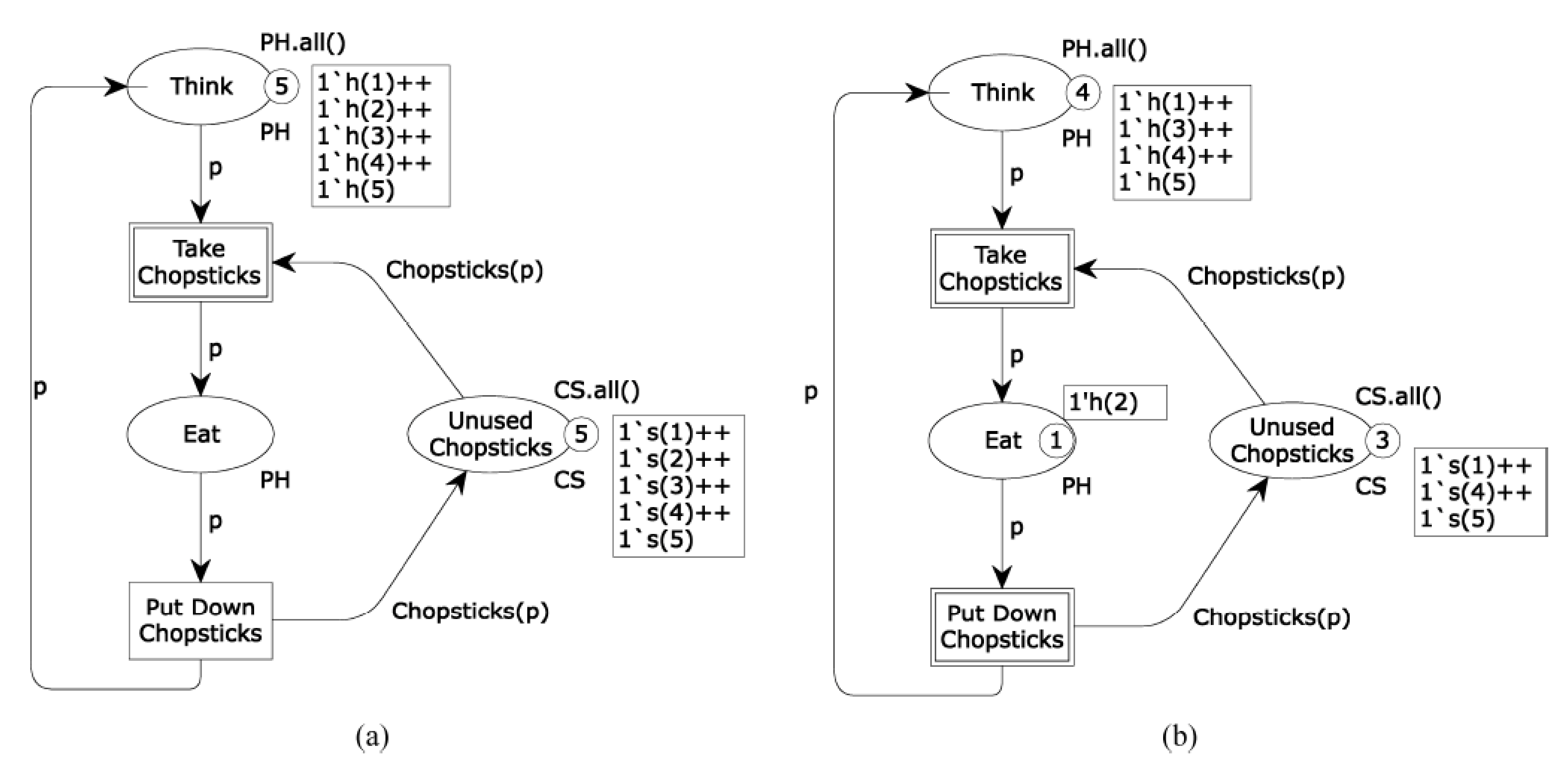


图 8-2　基本 Petri 网表示的哲学家就餐问题



(a)　　(b)

图 8-3　CPN 表示的哲学家就餐问题

的要简单，仅包含三个库所。其中一个库所表示思考的哲学家，一个表示就餐的哲学家，最后一个表示未使用的筷子。通过函数 Chopsticks 表示每一个哲学家只能使用其左边或者右边的筷子。而图 8-3(b)部分表示第二个哲学家开始吃饭时的情况，编号为 2 和 3 的筷子已经被使用，而处于思考状态的哲学家库所中少了一个编号为 2 的哲学家，而表示就餐状态的库所中多了一个编号为 2 的哲学家。

8.1.3 几种常见的系统结构模型

Petri 网具有十分丰富的结构描述能力，包括顺序、并行、冲突等。

顺序结构表示一系列系统行为具有明确的先后顺序关系，用 Petri 表示就是一个变迁必须在另一个变迁触发以后才能触发。图 8-4 给出了一个顺序结构模型的例子，在这个例子中包括 t_1，t_2 以及 t_3 三个变迁。变迁 t_2 只有在 t_1 被触发以后才会变成启用状态，同样 t_3 也只有在 t_2 触发以后，才变为启用状态，才可以触发。

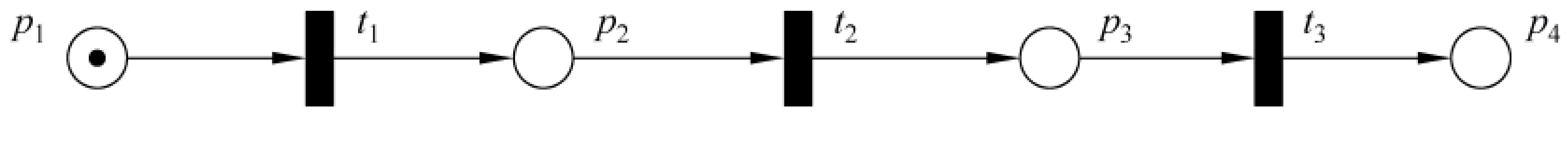


图 8-4 顺序结构模型

并行结构表示系统两个行为没有明确的时间顺序关系，在实际执行中，允许以不同的顺序执行，图 8-5 给出了一个并行结构的例子。

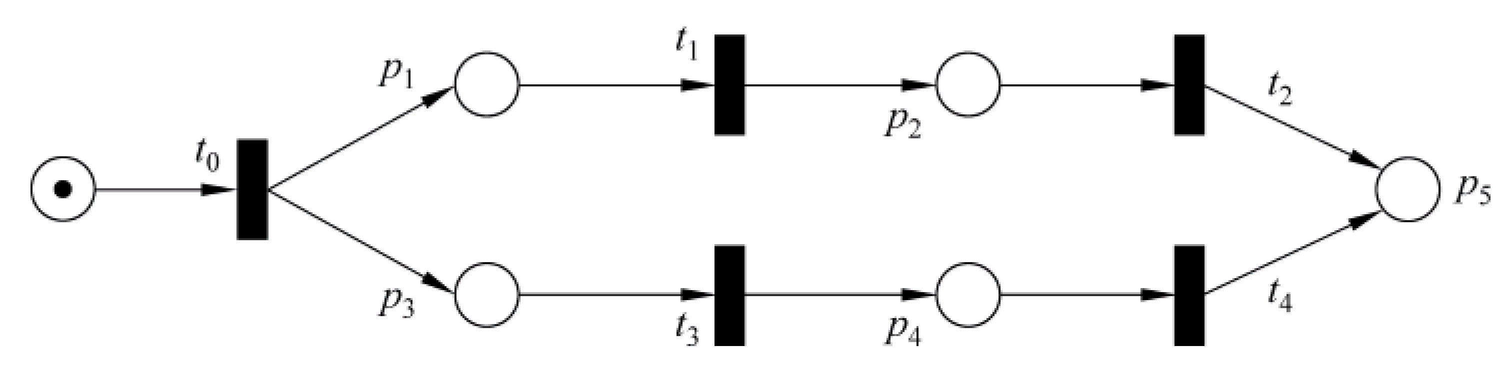


图 8-5 并行结构示意图

在图 8-5 中，t_0 执行以后，在 p_1 和 p_3 分别产生一个 Token。此时 t_1 和 t_3 同时处于启用状态，t_2 和 t_3 允许以不同的顺序触发。在这个图中，t_1 和 t_2 构成了顺序关系，同样地，t_3 和 t_4 一路构成顺序关系，但是在两路之间没有明确的时间约束关系。在这个图中，可以构成如下几种行为顺序。

(1) t_1,t_2,t_3,t_4

(2) t_3,t_4,t_1,t_2

(3) t_1,t_3,t_2,t_4

(4) t_1,t_3,t_4,t_2

(5) t_3,t_1,t_4,t_2

(6) t_3,t_1,t_2,t_4

这些行为顺序在实际系统中都是可以的，也是合法的。

冲突结构，指两个或者多个变迁都满足触发条件，但是在某个时刻只能有一个变迁会触发，具体哪一个触发，并没有特别的要求。冲突，又称为选择或者不确定图 8-6(a)部分

给出了一个冲突的示例，t_1 有两个输入弧分别来自 p_0 和 p_1，t_2 有两个输入弧分别来自 p_1 和 p_2，此时 t_1 和 t_2 都处于启用状态，但是当一个变迁触发以后，p_1 中的 Token 就被消耗，而另一个变迁将由启用状态变更为禁用状态。在一个时刻 t_1 和 t_2 只能被触发一个，若 t_1 被触发，那么 t_2 将处于禁用状态，同样当 t_2 被触发那么 t_1 将处于禁用状态。同样的道理，冲突也可以扩展到多路选择的情况，如图 8-6(b)所示。t_1、t_2、t_3 和 t_4 均处于启用状态，但是在某一个时刻只能有一个变迁被触发。在冲突结构中，只能在这么多路中间选择一路，不同路中的变迁不能同时触发。

对于图 8-6(a)允许的不同执行顺序为：

(1) t_1

(2) t_2

对于图 8-6(b)允许的不同执行顺序为：

(1) t_1

(2) t_2

(3) t_3

(4) t_4

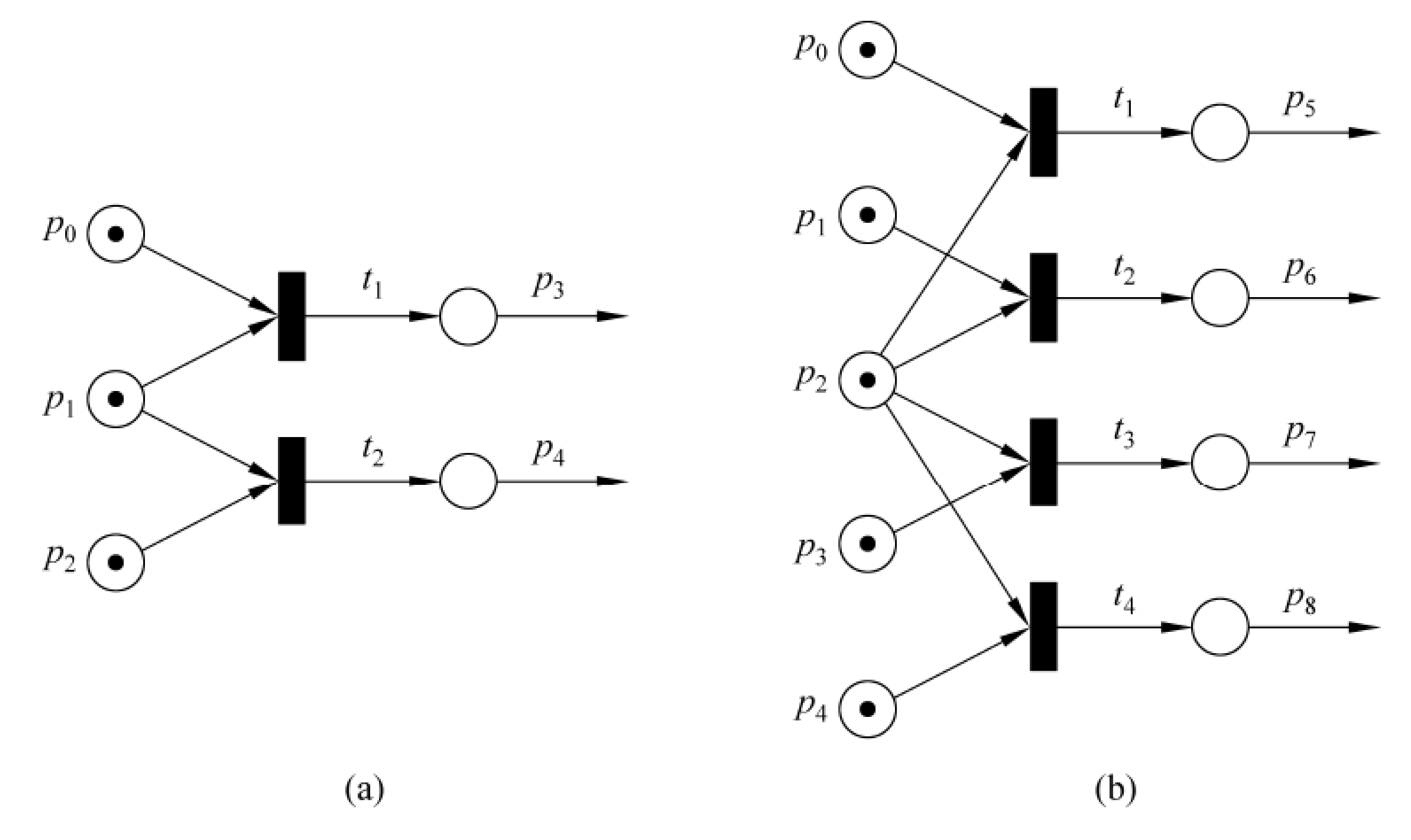


图 8-6　冲突结构

在实际系统中，除了上述单纯的顺序、并行、冲突结构外，也存在更加复杂的状态。如图 8-7 所示就是一个例子，在这个例子中，若 t_2 被触发，那么 t_1 和 t_2 将处于禁用状态。若 t_2 不触发，那么 t_1 和 t_3 将处于并行状态。这个系统行为，允许的执行顺序模式包括：

(1) t_2

(2) t_1，t_3

(3) t_3，t_1

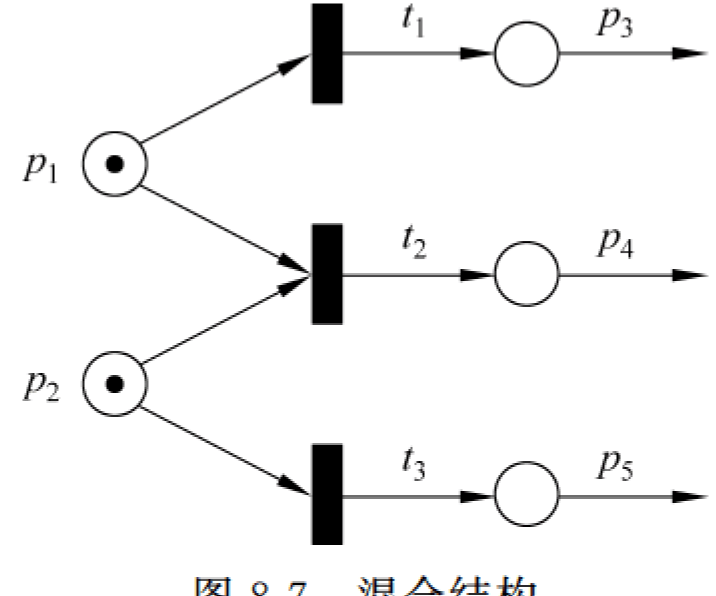


图 8-7　混合结构

例 8-2　一个简单的 Petri 网定义。

图 8-8 给出了一个 Petri 网示例，在这个网中：

$$p=\{p_1,p_2,p_3,p_4,p_5,p_6\}$$

$$T=\{t_1,t_2,t_3,t_4,t_5\}$$

$$F=\{(p_1,t_1),(t_1,p_2),(t_1,p_3),(p_2,t_2),(p_3,t_3),(t_2,p_4),\\(t_3,p_5),(p_4,t_4),(p_5,t_4),(t_4,p_6),(p_6,t_5),(t_5,p_1)\}$$

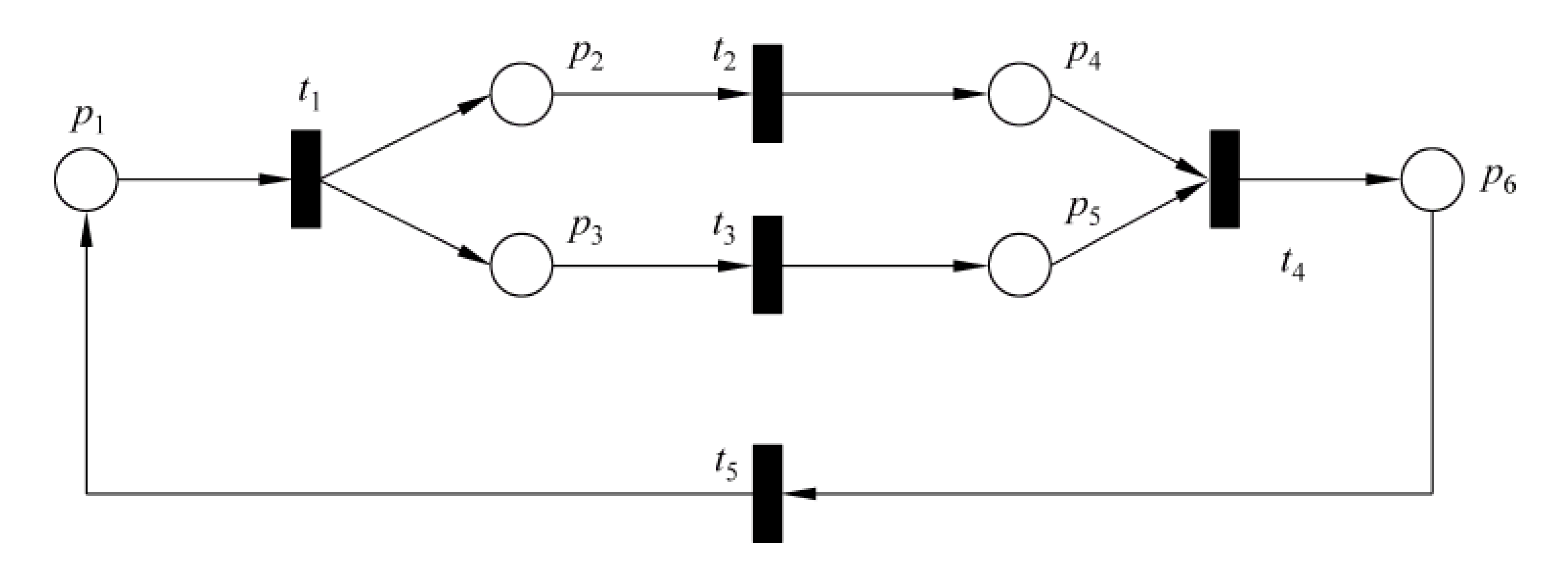


图 8-8　一个简单的 Petri 网示意图

在这个例子中，如果 t_1 被触发，那么将在 p_2 和 p_3 两个库所中分别产生一个 Token，意味着 t_2 和 t_3 两个变迁同时满足了触发条件。此时，对 t_2 和 t_3 两个变迁的触发时间没有特别的要求，既可以是先执行 t_2 后执行 t_3，也可以是先执行 t_3 后执行 t_2。这种模型表示了一种并行的系统模型。

8.1.4　Petri 网的行为性质

1. 可达性

可达性是 Petri 网行为性质的基础。若一个 Petri 网的初始标识为 M_0，在一系列的 $\sigma=t_1$、t_2、…、t_n 的触发作用下，网的状态从 M_0 转换成 M_n，则称 M_n 是从 M_0 可达的。记为：

$$M_0[\sigma>M_n$$

所有可达的标识组成一个可达标识集，记为：

$$R(N,M_0)\quad 或者\quad R(M_0)$$

从初始的标识 M_0 出发，在启用的变迁作用下，通过不同的迁移序列到达新的标识 M_1，M_1 在新的启用迁移作用下，又到达新的标识。如果将所有可以到达标识通过变迁连接起来构成的树，称为可达图。显然图的节点是由 M_0 经过启用的变迁而达到的标识。若对于图中的两个节点 M_1 和 M_2，若是 M_1 在变迁 t 的作用下，产生标识 M_2，即 $M_1[t>M_2$，那么 M_2 是 M_1 的后继节点，而 M_1 是 M_2 的前驱节点。

图 8-9(a)为一个 Petri 网，存在三个库所$\{p_0,p_1,p_2\}$和 4 个变迁$\{t_0,t_1,t_2,t_3\}$，其初始标识为 $M_0=\{1,0,0\}$，在初始标识下 t_0 和 t_1 分别处于启用状态。若触发 t_0，那么该网变迁为标识 $M_2=\{0,1,0\}$，若触发 t_1，网变迁为标识 $M_1=\{0,0,1\}$。若已经触发了 t_0，系统的标识为 M_2，此时只有 t_2 处于启用状态，当 t_2 触发系统的标识为 $M_1=\{0,0,1\}$。系统

的标识为 M_1，此时只有 t_3 处于启用状态，当 t_3 触发系统的标识为 $M_0=\{1,0,0\}$。由此 M_0，M_1，M_2 以及它们的变迁构成 Petri 网的可达图，如图 8-9(b)所示。

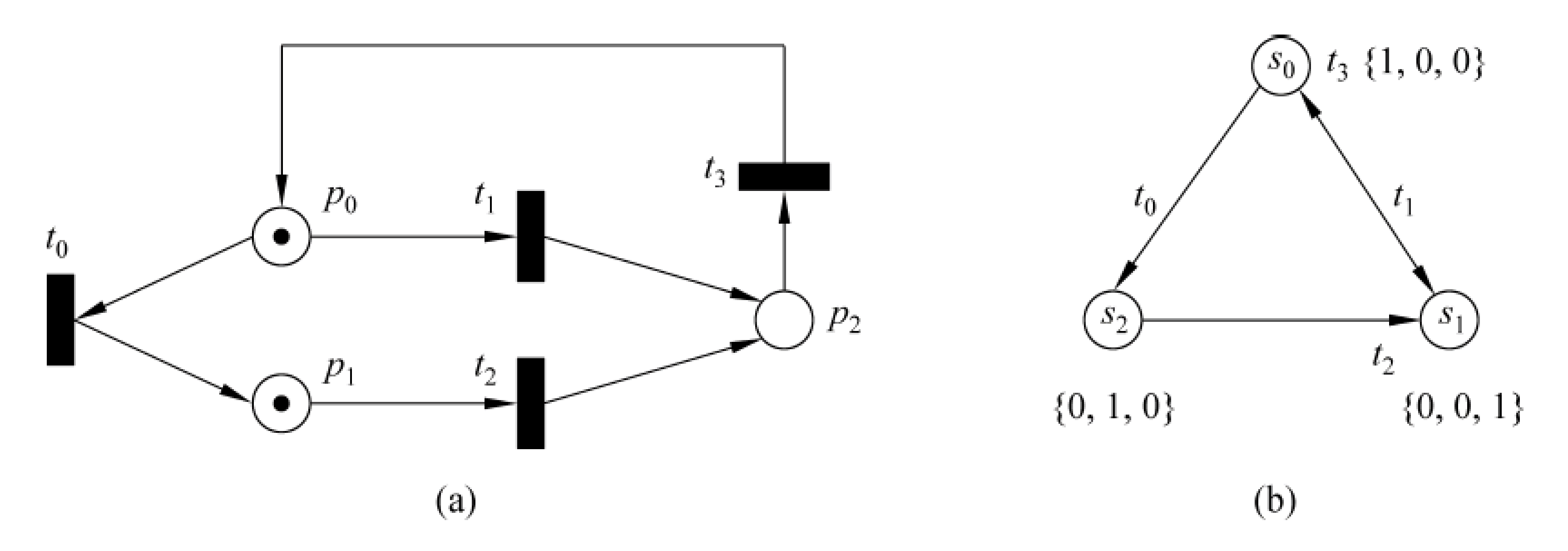


图 8-9　一个 Petri 网及其对应的可达图

2. 有界性和安全性

若一个 Petri 网 N 中，存在一个非负整数 K。从 M_0 开始的每一个可达标识的 Token 数量都不超过 K，那么称 N 为 K 有界。如果一个 Petri 网是 1 有界的，那么称该网是安全的。一个安全的 Petri 网每一个位置，要么有一个令牌，要么没有令牌。这里的 K 的定义和 Petri 网容量不同。这里的 K 实际无容量限制，但是在 Petri 网络的触发过程中，自动满足 K 上限。而容量是在每一个 P 上增加的限制。在图 8-10(a)中的 Petri 网，即使 p_1 和 p_2 没有容量的限制，但是所有库所包含的 Token 数量的最大值为 1，因此是安全的。而图 8-10(b)中的 p_3 随着 t_2 的不断触发，其中间的 Token 将不断增加，在没有容量设置的情况下，该网是无界的。

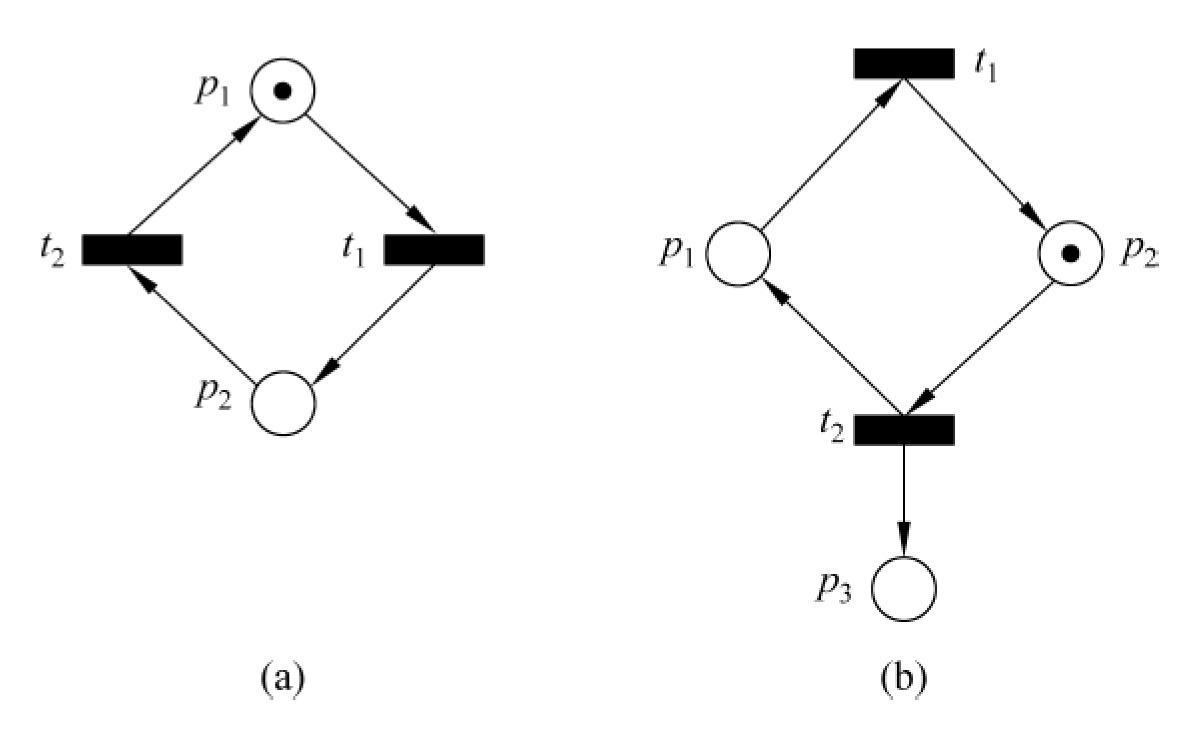


图 8-10　有界性和安全性

有界性反映了一个位置在系统运行过程中获得的最多的 Token 数量，除了和 Petri 网的结构有关，还和系统的初始标识的分布有关。有界性是 Petri 网一个重要的性质，确保系统在运行过程不会需要无限的资源。

3. 活性

一个系统潜在的问题就是死锁，如果一个系统存在死锁，那么处于死锁中的变迁永远

无法触发。

图 8-11 给出了一个死锁的例子。在这个 Petri 网中，t_1 的触发依赖于 p_4，而 p_4 中 Token 的获得依赖于t_2 的触发，而 t_2 的触发依赖于 p_3，而 p_3 中 Token 的获得依赖于 t_1 的触发。最后形成 t_1 的触发依赖于已经触发 t_1，这个条件永远也无法得到满足，从而 t_1 永远无法触发。同理可以分析，在这个网中，t_2 也永远无法触发，形成了一个典型的死锁场景。

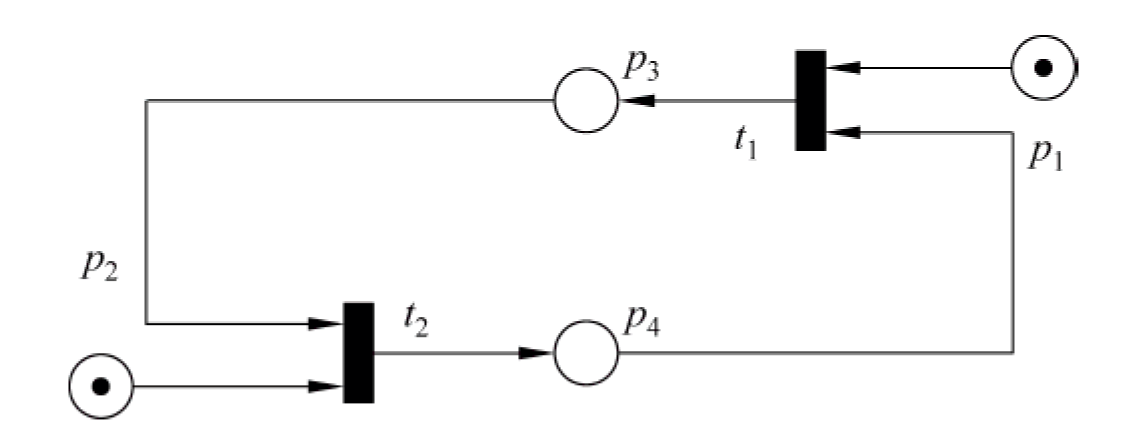


图 8-11　一个包含死锁的 Petri 网

在 Petri 网中，一个变迁可能处于以下几种情况。

L_0 级：t 在任何 $M=R(M_0)$情况下，都无法处于启用状态。

L_1 级：t 在某些 $M=R(M_0)$情况下，至少可以触发一次。

L_2 级：t 在某些 $M=R(M_0)$情况下，至少可以触发 k 次。

L_3 级：t 在某些 $M=R(M_0)$情况下，可以触发无限次。

L_4 级：t 在任何 $M=R(M_0)$情况下，都可以触发无限次。

实际上，处于 L_0 级的变迁是永远无法触发的。若 Petri 网中，从 M_0 可达的任意标识出发，都可以通过某一个迁移序列而最终触发 t，则称 t 在 M_0 下是活的(变迁 t 是 L_4 级活的)。如果所有的迁移都是活的，那么该 Petri 网是活的。如果一个系统是活的，那么这个系统必定不存在死锁。但是反过来，不存在死锁的系统不一定是活的，如图 8-12 所示。

4. 可逆性

在 Petri 网中，若满足：

$$\forall M \in R(M_0),\quad M_0 \in R(M)$$

则称该 Petri 网是可逆的。

换句话讲，对于一个可逆的 Petri 网，从任意一个可达的标识 M'出发，总存在触发序列 $\sigma=t_0,t_1,\cdots,t_n$使得该 Petri 网能够返回到初始状态标识 M_0。在很多实际系统中，并不一定要求系统的状态回到初始状态，只要求系统状态能够到达某一个特定状态即可。这个特定状态称为家状态(Home State)。若用 M_h 表示家状态，那么满足：

$$\forall M \in R(M_0),\quad M_h \in R(M)$$

8.1.5　基于 Petri 网的测试

一般而言，基于 Petri 网的测试都是基于状态可达图。

(1) 对系统进行 Petri 网建模,若已经存在 Petri 网模型,则直接进入步骤(2)。

(2) 验证 Petri 网模型的有界性、活性和可逆性。确保系统无死锁、存在家状态或者系统是可逆的。

(3) 建立 Petri 网的初始标识向量。

(4) 分析 Petri 网中的库所的输入矩阵、输出矩阵。

(5) 建立 Petri 网的可达图。

(6) 从初始标识出发,在可达图中依据不同的覆盖准则选择一条到家状态(或者初始状态)的路径。这里的覆盖准则包括边覆盖、节点覆盖等。

例 8-3 一个 Petri 网的测试。

图 8-13 中的 Petri 网有 6 个变迁 $T=\{t_0,t_1,t_2,t_3,t_4,t_5\}$,有 7 个库所。实际上,该网是一种简单的并行模式。t_0 和 t_1 构成一路行为,t_2 和 t_3 构成一路行为,两路行为可以并行。

在建立好该模型以后,接着要完成模型的验证工作。在 PIPE 中选择 State Space Analysis 执行分析,分析的结果为:有界、安全、无死锁。

其初始的标识向量为 $M_0=\{0,0,0,0,0,0,1\}$。

利用 PIPE 的 Incidence & Marking 模块得到 Petri 网的输入矩阵和输出矩阵,如表 8-3 和表 8-4 所示。

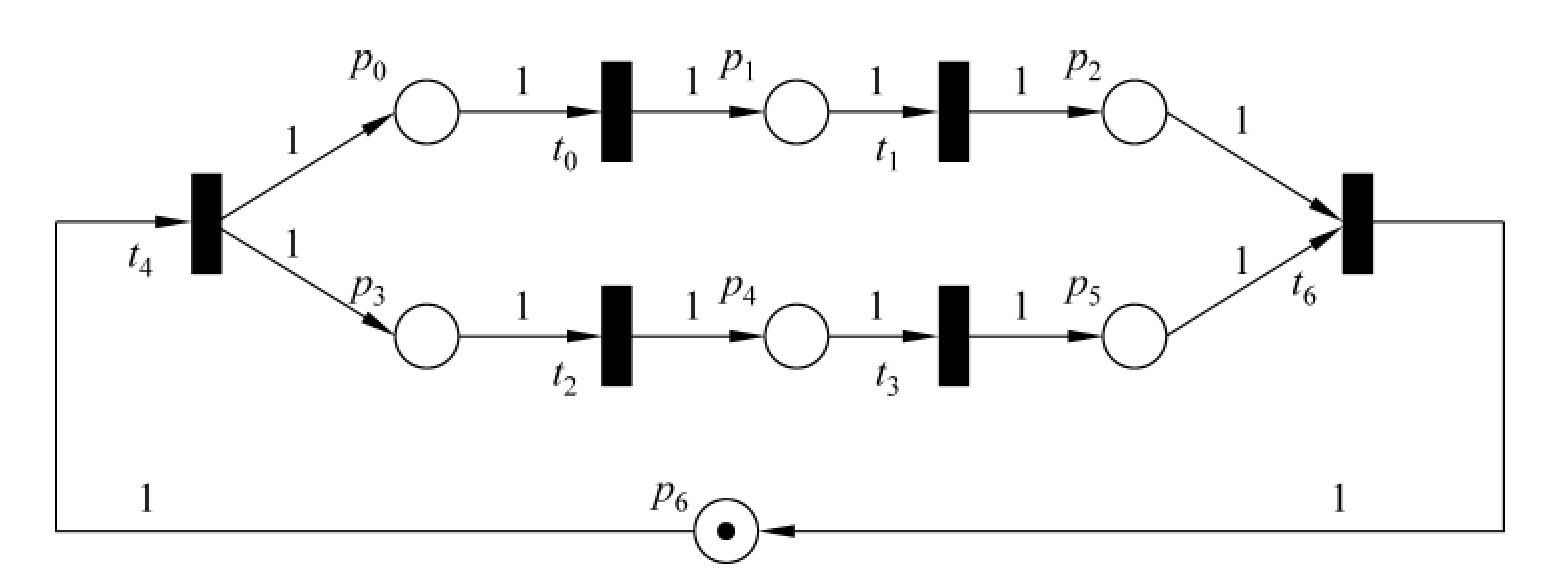


图 8-13 Petri 网测试示例模型

表 8-3 Petri 网的库所输入矩阵

	t_0	t_1	t_2	t_3	t_4	t_5
p_0	0	0	0	0	1	0
p_1	1	0	0	0	0	0
p_2	0	1	0	0	0	0
p_3	0	0	0	0	1	0
p_4	0	0	1	0	0	0
p_5	0	0	0	1	0	0
p_6	0	0	0	0	0	1

表 8-4 Petri 网的库所输出矩阵

	t_0	t_1	t_2	t_3	t_4	t_5
p_0	1	0	0	0	0	0
p_1	0	1	0	0	0	0
p_2	0	0	0	0	0	1
p_3	0	0	1	0	0	0
p_4	0	0	0	1	0	0
p_5	0	0	0	0	0	1
p_6	0	0	0	0	1	0

利用输入矩阵和输出矩阵，将其应用于 Petri 网产生可达图（或者直接利用 PIPE 的 Reachability/Coverability Graph 模块也可以产生可达图），如图 8-14 所示。

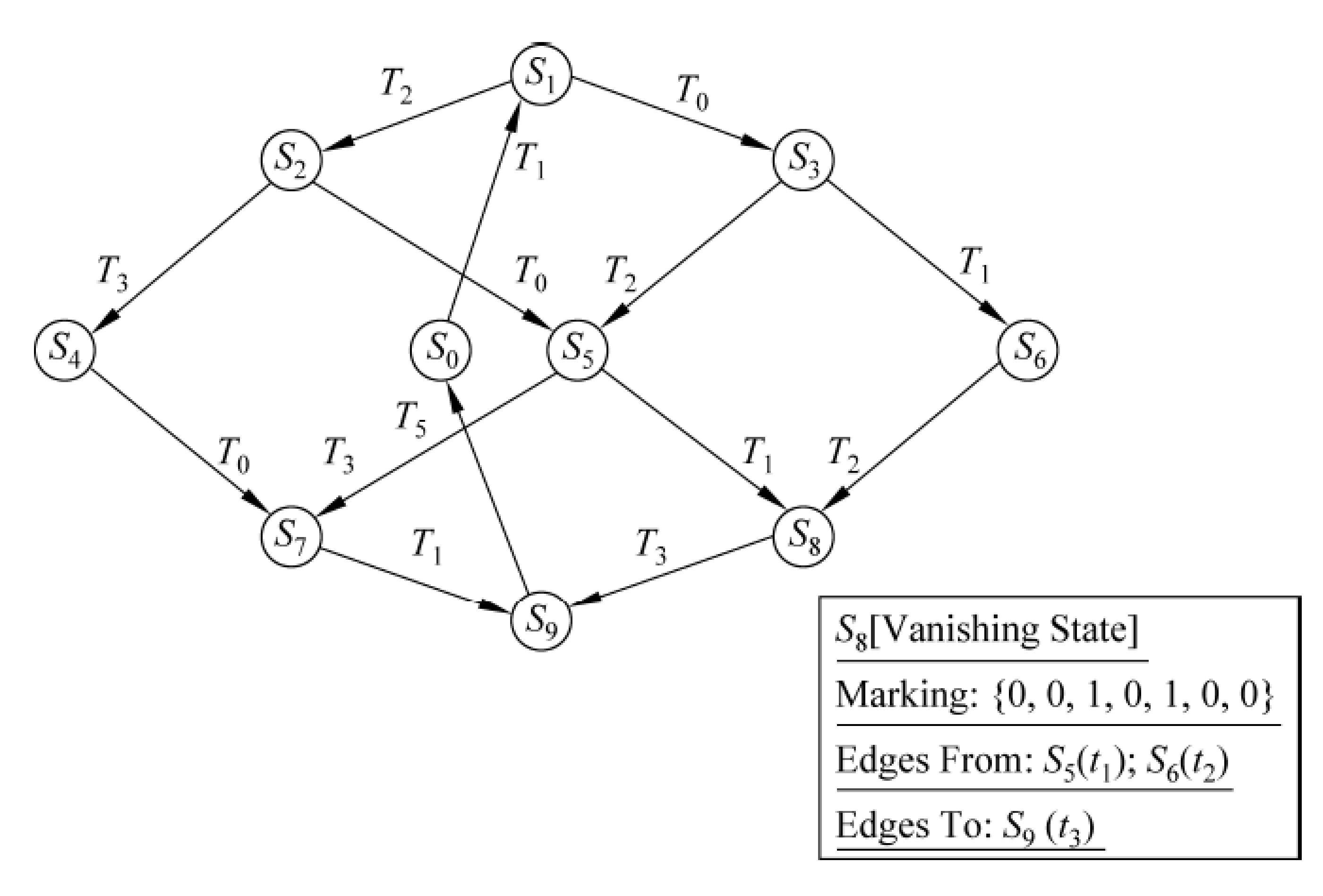


图 8-14 Petri 网对应的可达图

根据可达图，采用边覆盖准则产生从初始标识到初始标识的路径作为测试路径，如表 8-5 所示。

表 8-5 测试路径

序号	可达图的路径	Petri 变迁触发顺序
1	$S_0 T_4 S_1 T_0 S_3 T_1 S_6 T_2 S_8 T_3 S_9 T_5 S_0$	$T_4 T_0 T_1 T_2 T_3 T_5$
2	$S_0 T_4 S_1 T_0 S_3 T_2 S_5 T_1 S_8 T_3 S_9 T_5 S_0$	$T_4 T_0 T_2 T_1 T_3 T_5$
3	$S_0 T_4 S_1 T_0 S_3 T_2 S_5 T_3 S_7 T_1 S_9 T_5 S_0$	$T_4 T_0 T_2 T_3 T_1 T_5$
4	$S_0 T_4 S_1 T_2 S_2 T_0 S_5 T_1 S_8 T_3 S_9 T_5 S_0$	$T_4 T_2 T_0 T_1 T_3 T_5$
5	$S_0 T_4 S_1 T_2 S_2 T_0 S_5 T_3 S_7 T_1 S_9 T_5 S_0$	$T_4 T_2 T_0 T_3 T_1 T_5$
6	$S_0 T_4 S_1 T_2 S_2 T_3 S_4 T_0 S_7 T_1 S_9 T_5 S_0$	$T_4 T_2 T_3 T_0 T_1 T_5$

活性等。

```
Report generated: Tue Aug 18 21:20:20 2015

Statistics
------------------------------------------------------------
  State Space
     Nodes:   11
     Arcs:    33
     Secs:    0
     Status: Full

  Scc Graph
     Nodes:   8
     Arcs:    9
     Secs:    0

Boundedness Properties
------------------------------------------------------------

  Best Integer Bounds
                                       Upper       Lower
     Page1'BkBtn1                      1           1
     Page1'Coin_Cache1                 1           1
     Page1'Coin_Insert_Slot 1          1           1
     Page1'Coin_Pool 1                 1           1
     Page1'Coin_Slot 1                 1           1
     Page1'DrkBtn 1                    1           1
     Page1'Drk_Pool 1                  1           1
     Page1'Drk_Slot 1                  1           1
     Page1'Info_Window 1               1           1

  Best Upper Multi-set Bounds
     Page1'BkBtn 1              1`DOWN
     Page1'Coin_Cache 1         1`(0,0)++1`(0,2)
     Page1'Coin_Insert_Slot 1
                                1`(0,2)
     Page1'Coin_Pool 1          1`(0,2)++1`(1,0)
     Page1'Coin_Slot 1          1`(0,0)++1`(0,2)++1`(1,0)
     Page1'DrkBtn 1             1`DOWN
     Page1'Drk_Pool 1           1`4++1`5
     Page1'Drk_Slot 1           1`0
     Page1'Info_Window 1        1`Ready++1`NoChange
```

```
  Best Lower Multi-set Bounds
     Page1'BkBtn 1            1`DOWN
     Page1'Coin_Cache 1       empty
     Page1'Coin_Insert_Slot 1
                              1`(0,2)
     Page1'Coin_Pool 1        empty
     Page1'Coin_Slot 1        empty
     Page1'DrkBtn 1           1`DOWN
     Page1'Drk_Pool 1         empty
     Page1'Drk_Slot 1         1`0
     Page1'Info_Window 1      empty

Home Properties
-------------------------------------------------------------------------
  Home Markings
     [10,11]

Liveness Properties
-------------------------------------------------------------------------
  Dead Markings
     None

  Dead Transition Instances
     None

  Live Transition Instances
     All
```

依据 CPN 模型测试覆盖准则来讨论基于 CPN 状态空间的测试方法，对于自动饮料机而言，主要包括下面 4 个场景。

场景 1：用户仅有一枚价值 5 毛的硬币。

在该场景下，用户无法成功购买饮料，场景的重点在于测试不同操作次序产生的系统状态是否正确。

场景 2：用户用足够的钱购买饮料，并且机器有足够的零钱。

假设用户拥有两枚 1 元的硬币，这个场景中库所 Coin_Insert_Slot 的初始标记为 (0,2)，而库所 Drk_Pool 和 Coin_Pool 的初始标记为 1`5 和 1`(5,0)，分别表示饮料和零钱足够。

场景 3：用户需要找零，但饮料机中没有零钱。

Coin_Pool 的初始状态设置成为 1`(0,0)，表示机器中没有零钱。由于没有零钱可以找零，在用户需要找零时是无法成功购买饮料的。为了减少用户金额变化而产生的状态数量，防止出现状态爆炸，将 Insert coin 变迁的弧设置成双向弧。

场景 4：机器中的饮料已经售完。

表 8-6　场景 1 中 满足 SC 准则的测试集

状态编号	前驱节点	变迁序列	状态编号	前驱节点	变迁序列
1	null	Initial marking	3	2	Push_DrkBtn
2	1	Insert_Coin	4	3	Push_BkBtn

在每一条变迁信息中，紧跟着变迁序号的是变迁的起始状态和终止节点。在一些特殊的变迁中，起始节点和终止节点都是该节点本身。例如，在用户没有塞入硬币之前，用户按下“饮料”按钮 DrkBtn 和“退币”按钮 BkBtn 均不应引起系统状态的改变，前面讨论的第二条和第三条的变迁信息反映的就是这种情况，即状态节点 1 中的变迁 Push_DrkBtn 和 Push_BkBtn 不会引起状态的改变，其起始状态和终止状态都是状态节点 1。在变迁覆盖中，无论变迁是否指向同一节点均应被测试用例所覆盖，通过变迁覆盖 TC 产生的测试集如表 8-7 所示。

表 8-7　场景 1 中满足 TC 准则的测试集

变迁编号	起始节点	终止节点	变 迁 序 列
1	1	1	Push_DrkBtn
2	1	1	Push_BkBtn
3	1	2	Insert_Coin
4	2	3	Push_DrkBtn
5	2	4	Push_BkBtn
6	3	3	Push_DrkBtn
7	3	4	Push_BkBtn
8	4	4	Push_DrkBtn
9	4	4	Push_BkBtn

状态对由当前状态节点的两个直接的相邻节点组成。如节点 3 由两个前驱状态节点和两个后继节点构成。针对节点 3 显然可以构造满足 SPC 准则的测试集，如表 8-8 所示。

表 8-8　场景 1 节点 3 满足状态对覆盖的测试集

编号	覆盖的状态对	变 迁 序 列
1	2，3，3	Push_DrkBtn Push_DrkBtn
2	2，3，4	Push_DrkBtn Push_BkBtn
3	3，3，3	Push_DrkBtn Push_DrkBtn
4	3，3，4	Push_DrkBtn Push_BkBtn

满足 SPC 准则的测试集数量 TCN 可以通过下面的公式进行计算。

8.2.2 蜕变测试的基本理论

在实际测试过程中，软件在一定的输入下产生了一个实际的运行结果，有时尽管无法直接判断这些实际运行结果，但是人们发现这些输出结果之间存在一定的约束关系，这些约束关系可以通过程序的规格说明书推导出来。软件实际运行结果的约束关系，称为蜕变关系。

假设在没有任何计算器或者直接的数学函数库的情况下，程序 P 实现一个函数 $\cos(x)$，除了特殊的角度例如 30°、45°、60°等以外，对于一个任意的角度，要判断其预期的输出是非常困难的。余弦函数具有如下性质。

$$\cos(-x) = \cos(x)$$

$$\cos(\pi - x) = \cos(x)$$

$$\cos(\pi + x) = -\cos(x)$$

对于程序 P 而言，这些关系就是蜕变关系。在一次实际执行时，如果输入 I_1 产生实际输出为 O_1，产生的初始关系为 $\cos(I_1)=O_1$。在程序正确的前提下，显然必须满足如下关系。

$$\cos(-I_1) = \cos(I_1) = O_1$$

$$\cos(\pi - I_1) = \cos(I_1) = O_1$$

$$\cos(\pi + I_1) = -\cos(I_1) = -O_1$$

$$\cos(2\pi + I_1) = \cos(I_1) = O_1$$

蜕变测试由 T. Y Chen 等人提出，利用程序执行结果之间的关系来测试程序，无须构造预期输出。为了区分蜕变测试用例和用于构造蜕变测试的测试用例，将用于构造蜕变测试用例的用例，称为原始测试用例。

若程序 P 实现函数 f，那么程序 P 必须遵守函数 f 的约束条件，否则程序 P 是不正确的。

若程序 P 是函数 f 的一种实现，$x_1,x_2,\cdots,x_n(n>1)$ 是 f 的 n 组变元，这些变元对应的函数运行结果分别为 $f(x_1),f(x_2),\cdots,f(x_n)$，若输入变元 $x_1,x_2,\cdots,x_n$ 之间满足关系 r，那么 $f(x_1),f(x_2),\cdots,f(x_n)$满足关系 r_f，即：

$$r(x_1,x_2,\cdots,x_n) \Rightarrow r_f(f(x_1),f(x_2),\cdots,f(x_n))$$

则将由输入关系和输出关系构成的二元组(r,r_f)称为 P 的一个蜕变关系。

例如，若程序 $P_{\cos}$ 实现了上述的余弦函数 $\cos(x)$，若其输入变元 x_1,x_2 满足关系：

$$r: x_1 + x_2 = 0$$

那么余弦函数的值必须满足关系：

$$r_f: \cos(x_1) = \cos(x_2)$$

显然，二元组：

$$(r: x_1 + x_2 = 0, r_f: \cos(x_1) = \cos(x_2))$$

是程序 P 的一个蜕变关系。

同样可以知道，若根据前面的分析，下面的一些关系也是程序 $P_{\cos}$ 的蜕变关系。

$$(r: x_1 + x_2 = \pi, r_f: \cos(x_1) = \cos(x_2))$$

$$(r: x_2 - x_2 = \pi, r_f: \cos(x_1) = -\cos(x_2))$$

若 $I_1, I_2, \cdots, I_n$ 是 $x_1, x_2, \cdots, x_n$ 的实际取值，而 $P_{\cos}(x_1), P_{\cos}(x_2), \cdots, P_{\cos}(x_n)$ 分别是程序 $P_{\cos}$ 在对应输入下的实际输出。若程序 $P_{\cos}$ 是正确的，必须满足关系：

$$r(I_1, I_2, \cdots, I_n) \Rightarrow r_f(P_{\cos}(I_1), P_{\cos}(I_2), \cdots, P_{\cos}(I_n))$$

显然对于程序 P，如果输入为 38°，程序 $P_{\cos}$ 运行的结果为：

$$P_{\cos}(-38°) = 0.788$$

可以同时依据上述的蜕变关系进行测试：

$$P_{\cos}(\pi - 38°) = P_{\cos}(142°) = -0.788$$

$$P_{\cos}(\pi + 38°) = P_{\cos}(-218°) = -0.788$$

$$P_{\cos}(2\pi + 38°) = P_{\cos}(398°) = 0.788$$

从上述例子可以看出，一个程序 $P_{\cos}$ 其蜕变关系不止一个。

同样，蜕变关系的输入变元可以不只是两个，例如：

$$\sin x + \sin y + \sin z - \sin(x + y + z) = 4\sin\frac{(x+y)}{2} \times \sin\frac{(x+z)}{2} \times \sin\frac{(y+z)}{2}$$

蜕变关系中，也可能不是特别明显，例如：

$$\sin(3 \times x) = 3 \times \sin(x) - 4\sin3(x)$$

$$\sin(5 \times x) = 16 \times \sin^5(x) + 5 \times \sin(3 \times x) - 10 \times \sin(x)$$

假设用 $R_i = (r_i, R_i)$ 表示第 i 个蜕变关系，那么：

$$R = (R_1, R_2, \cdots, R_n)$$

表示程序 P 的蜕变关系集合。

8.2.3 蜕变测试的过程

一个蜕变测试的过程通常包含 4 个步骤，包括初始测试用例选取、执行初始测试用例、利用蜕变关系集并构造衍生测试用例、检查原始用例和衍生用例是否满足蜕变关系，如图 8-17 所示。

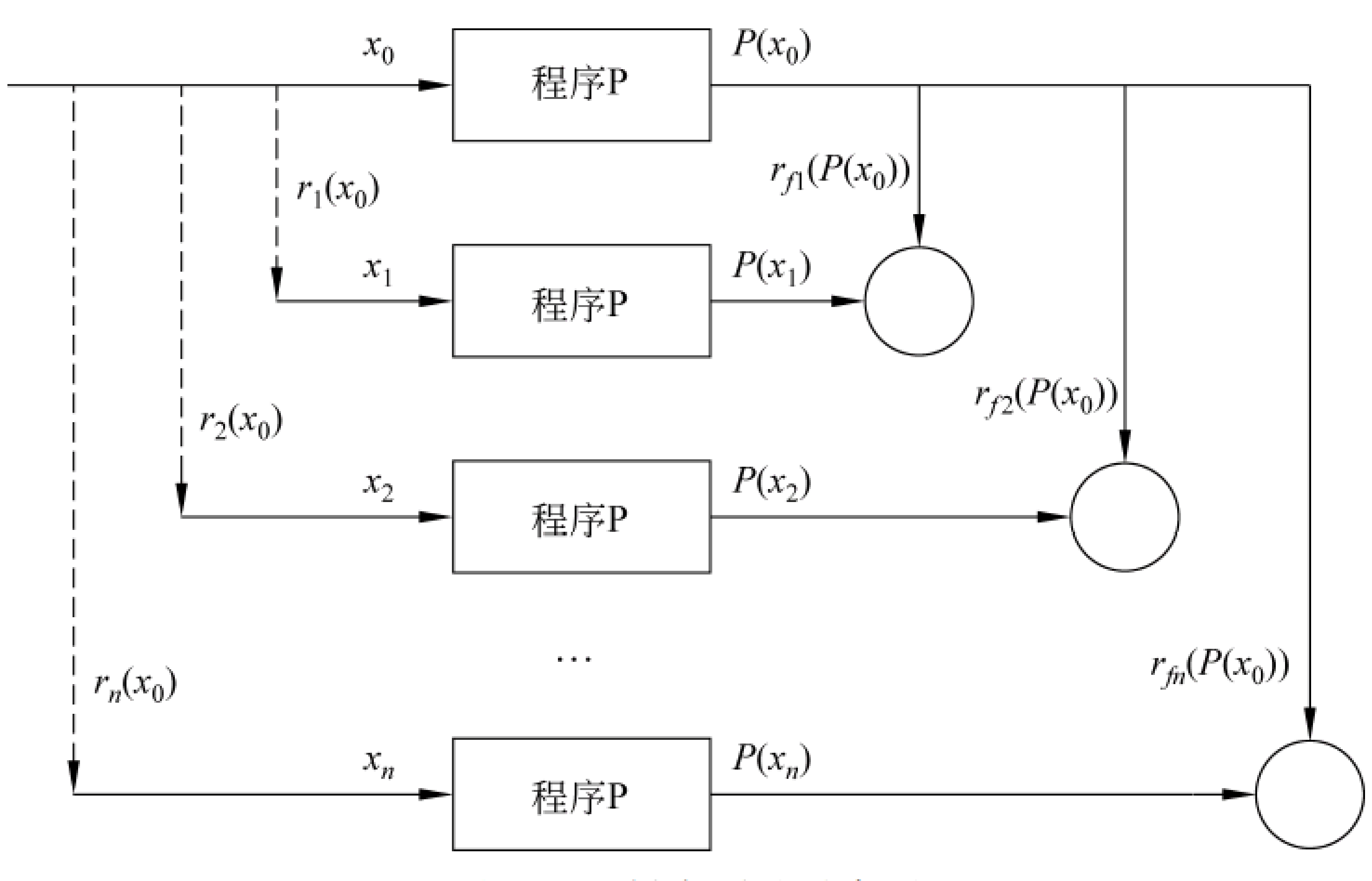


图 8-17　蜕变测试示意图

```
    lcs0= lcs(a0, b0)
    lcs1= lcs(a1,b0)
    lcs2= lcs(a0, b1)
    flcs= rf1(lcs1, lcs2)
    plcs= lcs0
    print 'lcs(a0,b0)=',lcs0,',lcs(a1,b1)=',plcs, ',',
    if plcs==flcs:
        print 'pass'
    else:
        print 'false'

mt1()

######################
def r2(a,b):
    a1= 'E'+a #E和 F不在 a 和 b中
    b1= 'F'+b
    return (a1,b1)

def rf2(lcs):
    return lcs

def mt2():
    global a0,b0
    print "mt2:",
    lcs0= lcs(a0, b0)
    (a1,b1)= r2(a0, b0)
    lcs1= lcs(a1, b1)
    plcs= lcs0
    flcs= rf2(lcs1)
    print 'lcs(a0,b0)=',lcs0,',lcs(a1,b1)=',plcs, ',',
    if plcs==flcs:
        print 'pass'
    else:
        print 'false'

mt2()
######################
def r3(a,b):
    a1=a+ 'E'   #E和 F不在 a 和 b中
    b1=b+ 'F'
    return (a1,b1)

def rf3(lcs):
```

```
    return lcs

def mt3():
    global a0,b0
    print "mt4:",
    lcs0= lcs(a0, b0)
    (a1,b1)= r3(a0, b0)
    lcs1= lcs(a1, b1)
    plcs= lcs0
    flcs= rf3(lcs1)
    print 'lcs(a0,b0)=',lcs0,',lcs(a1,b1)=',plcs, ',',
    if plcs==flcs:
        print 'pass'
    else:
        print 'false'

mt3()
#####################
def r4(a,b):
    a1= 'E'+a
    b1= 'E'+b
    return (a1,b1)

def rf4(lcs):
    return 'E'+lcs

def mt4():
    global a0,b0
    print "mt4:",
    lcs0= lcs(a0, b0)
    (a1,b1)= r4(a0, b0)
    plcs= lcs(a1, b1)
    flcs= rf4(lcs0)
    print 'lcs(a0,b0)=',lcs0,',lcs(a1,b1)=',plcs, ',',
    if plcs==flcs:
        print 'pass'
    else:
        print 'false'

mt4()
#####################
def r5(a,b):
    a1=a+ 'E'
    b1=b+ 'E'
```

```
    return (a1,b1)

def rf5(lcs):
    return lcs+ 'E'

def mt5():
    global a0,b0
    print "mt5:",
    lcs0= lcs(a0, b0)
    (a1,b1)= r5(a0, b0)
    plcs= lcs(a1, b1)
    flcs= rf5(lcs0)
    print 'lcs(a0,b0)= ',lcs0,',lcs(a1,b1)= ',plcs, ',',
    if plcs==flcs:
        print 'pass'
    else:
        print 'false'

mt5()
######################
def r6(a,b):
    return (b,a)

def rf6(plcs,flcs):
    return len(plcs),len(flcs)

def mt6():
    global a0,b0
    print "mt6:",
    lcs0= lcs(a0, b0)
    (a1,b1)= r6(a0, b0)
    lcs1= lcs(a1, b1)
    (lplcs,lflcs)= rf6(lcs0,lcs1)
    print 'lcs(a0,b0)= ',lcs0,',lcs(a1,b1)= ',lcs1, ',',
    if lplcs==lflcs:
        print 'len is  equal,pass'
    else:
        print 'len is not equal,false'

mt6()
######################
def r7(a,b):
    a1= a[::-1]
    b1= b[::-1]
```

```
    return a1,b1

def rf7(plcs,flcs):
    return len(plcs),len(flcs)

def mt7():
    global a0,b0
    print "mt7:",
    lcs0= lcs(a0, b0)
    (a1,b1)= r7(a0, b0)
    lcs1= lcs(a1, b1)
    (lplcs,lflcs)= rf7(lcs0,lcs1)
    print 'lcs(a0,b0)= ',lcs0,',lcs(a1,b1)= ',lcs1, ',',
    if lplcs= = lflcs:
        print 'len is  equal,pass'
    else:
        print 'len is not equal,false'

mt7()
```

8.3 基于变异的软件测试方法

基于变异的软件测试方法,通过将缺陷引入程序生成变异体,再将测试数据集放到错误的程序中运行,期望可以发现这些缺陷。发现的缺陷越多,则说明测试用例越完备。同时,变异测试还可以评估软件中存在的缺陷情况。

8.3.1 变异测试的概念

变异测试中引入缺陷的过程就是"变异",缺陷注入的规则是"变异算子",包含变异算子的程序是"变异体"。表 8-11 是一个变异算子示例,该变异算子将语句 if(a+b>2)中的符号">"变为">=",原程序变成变异体,整个过程是一个变异过程。

表 8-11 一个变异算子示例

原程序	变异体
… if(a+b>2): return True return False	… if(a+b>=2): return True return False

变异只改变原程序的一小部分,可能是一行程序或者几行程序,大部分的程序是没有改变的。根据执行变异算子的次数,变异可分为一阶变异和高阶变异。一阶变异是指在原程序上执行单一的变异算子形成变异体,高级变异是在原程序上执行多个变异算子形

成变异体。相对于一阶变异,高阶变异一般更容易被发现。表 8-11 中采用的是一阶变异,只使用了一个变异算子。

原程序变异后,将测试数据放到变异体和原程序中运行,可以得到两者的结果。如果变异体和原程序的结果不同,则说明程序发现了变异体,变异体被杀死;如果结果相同,则说明测试用例没有发现变异体,需要用其他的测试数据进行测试。针对表 8-11 中的变异体,如果测试用例的输入是 $a=2,b=3$,则两者的运行结果都返回 True,结果是一致的,无法杀死变异体。但如果将输入换成 $a=1,b=1$,则原程序和变异体的结果不同,原程序返回 False,变异体返回 True,变异体被杀死。

理想的情况下,所有的变异体都能被杀死,则测试用例是充分的。但是实际测试中很难实现杀死 100%的变异体,可以设置要求达到的比率。变异测试使用杀死非等价变异体的比率来判定测试用例的完备性。

下面是变异测试的一些基本概念。

(1) 被杀死变异体:存在某个测试用例,变异体的执行结果和原程序的执行结果不一致,该变异体被杀死。

(2) 活变异体:对于所有的测试用例,变异体的执行结果和原程序的执行结果都一致,则变异体是活变异体,没有被杀死。

(3) 等价变异体:如果变异体和原程序语法上不同,但是语义上相同,则该变异体是原程序的等价变异体。

(4) 变异评分:用来衡量测试用例集的测试充分性,公式如下:

$$\mathrm{KM} = \frac{K}{M-E}$$

其中,KM 为变异评分,即杀死非等价变异体的比率,K 是被杀死的变异体的数量,M 是全部变异体的数量,E 是原程序中等价变异体的数量。

在变异测试中,KM 的取值越大,则说明测试用例越完备。如果所有的测试用例都已经测试过了,但仍有变异体没有被杀死,则只能是下面两种原因。

(1) 该变异体是原程序的等价变异体,和原程序语法上不同,但是语义上相同,则测试用例必然不能杀死该变异体,它的执行结果和原程序的执行结果是一致的。

(2) 该变异体是可以被杀死的,但是由于测试用例不够完备,导致无法检测到该变异体。在这样的情况下,应当增加测试用例来尝试杀死变异体,达到测试的要求。

因此变异测试使用的 KM 来说明测试用例的完备性,排除了等价变异体的干扰。

8.3.2 变异算子

变异测试中,变异体是由原程序经过变异算子变异得到的。变异算子模拟了软件程序中的一些错误,因此变异算子的设计将直接影响到变异测试的效率和结果。由于程序设计语言不同,使用的变异算子也会不同。针对各个程序设计语言的特点来设计相应的变异算子,可以更好地评估测试数据集。

传统的变异算子中，变异算子 ABS、ROR 和 AOR 比较常用。以 ROR 为例，Python 的关系运算符有 7 种：＝＝、！＝、＜＞、＞、＜、＞＝和＜＝，其中，“＝＝”和“＜＞”是等价的。ROR 变异算子就是将程序中的一个关系运算符用其他关系运算符代替，生成变异体进行测试。现有一个程序用于比较两个变量是否相等，采用 ROR 变异算子设计变异体，原程序和变异体的对比如表 8-13 所示。

表 8-13　比较两个变量是否相等的 ROR 变异算子

变异算子	原　程　序	变　异　体
ROR1	def isequal(a,b)： if a！＝b： print "a is not equal to b" else： print "a is equal to b"	def isequal(a,b)： ➡ if a＝＝b： print "a is not equal to b" else： print "a is equal to b"
ROR2	def isequal(a,b)： if a！＝b： print "a is not equal to b" else： print "a is equal to b"	def isequal(a,b)： ➡ if a＜＞b： print "a is not equal to b" else： print "a is equal to b"
ROR3	def isequal(a,b)： if a！＝b： print "a is not equal to b" else： print "a is equal to b"	def isequal(a,b)： ➡ if a＞b： print "a is not equal to b" else： print "a is equal to b"
ROR4	def isequal(a,b)： if a！＝b： print "a is not equal to b" else： print "a is equal to b"	def isequal(a,b)： ➡ if a＜b： print "a is not equal to b" else： print "a is equal to b"
ROR5	def isequal(a,b)： if a！＝b： print "a is not equal to b" else： print "a is equal to b"	def isequal(a,b)： ➡ if a＞＝b： print "a is not equal to b" else： print "a is equal to b"
ROR6	def isequal(a,b)： if a！＝b： print "a is not equal to b" else： print "a is equal to b"	def isequal(a,b)： ➡ if a＜＝b： print "a is not equal to b" else： print "a is equal to b"

表 8-13 中有 6 个 ROR 变异体，主要是将语句“if a！＝b”中的“！＝”转换为其他 6 种关系运算符。因为“！＝”和“＜＞”语法不同但语义相同，因此变异体 ROR2 是一个等价变异体，其他都是非等价变异体。

表 8-15 OMR 变异

原 程 序	变 异 体

class Stack{
    …
    void push(int e){…}
    void push(int e,int n){
        …
    }
    …
}
	class Stack{
	    …
	    void push(int e){…}
	    void push(int e,int n){
	➡ this,add(e);
	    }
	    …
	}

该程序中有两个 push 函数,一个有一个参数,另一个有两个参数。变异算子 OMR 将有两个参数的函数替换成了一个参数的函数。这种变异是 Java 中可能会存在的缺陷,但是在 Python 中却不会存在这种缺陷,因为 Python 不支持多个相同名称的函数,也就不会存在冗余。如果 Python 中有多个名称相同的方法,则下面的方法会自动覆盖上面的方法。

下面用 Python 语言对部分适用的类变异算子进行说明。

1. AMC(修改访问权限)

AMC 变异算子改变了原来程序中变量或者方法的访问权限,模拟的是原程序可能存在的访问权限的缺陷设置。该变异体可以引导测试人员设计测试用例找出程序中访问权限的缺陷。Python 语言中变量和方法的访问权限有两种,一种是公开的,另一种是私有的。通常访问权限不符合要求有两种情况:原本需要公开的变量或方法现在是私有的,无法在类的外部调用;或者原本私有的变量现在是公开的,可以在类的外部调用。表 8-16 是一个简单的例子,变异体 AMC 将原程序中私有的变量改为公开的变量。

表 8-16 AMC 变异算子

原 程 序	变 异 体
class List: __size=0 …	class List: ➡ size=0 …

2. IHI(插入隐藏变量)

IHI 变异算子是针对类的继承特性来说的。子类继承了父类的特性,如果需要调用父类变量,可以直接调用父类里的变量,不需要重新定义。IHI 变异算子在子类中重新定义了父类的变量,导致实际访问时无法访问父类的变量,只能访问子类的变量。这种变异算子只有当子类定义的变量有缺陷时才会被发现。表 8-17 是一个简单的例子。

5. IOP（改变重写方法调用位置）

子类重写父类的方法后，需要对父类的方法进行修改。实际情况中，子类可能仍然需要在重写的方法中调用被重写的父类方法，并在这个基础上对该方法增加一些其他的内容。父类方法的调用可能是在开头也可能在结尾处，或者是一句语句的上面或者下面。IOP 变异算子改变了子类中调用父类的位置，可能导致错误的输出。表 8-20 是一个简单的例子。

表 8-20　IOP 变异算子

原　程　序	变　异　体
class List(object)： size=0 … def SetEnv(self)： self. size=5 … class Stack(List)： … def SetEnv(self)： super(Stack,self). SetEnv() self. size=10	class List(object)： size=0 … def SetEnv(self)： self. size=5 … class Stack(List)： … def SetEnv(self)： ➡ self. size=10 ➡ super(Stack,self). SetEnv()

6. IOR（重新命名重写方法）

IOR 变异算子是为了检测子类重写的方法是否对父类的其他方法产生影响。假设父类有两个方法 f()和 m()，其中，方法 m()需要调用方法 f()。现在子类重写了方法 f()，则子类的实例在调用 m()方法的过程中调用的是子类的 f()而不是父类的 f()。如果现在修改了父类的方法 f()的名称为 f1()，且父类的方法 m()中的 f()的名称也改为 f1()，则子类的实例在调用 m()方法的过程中调用的是父类的 f1()方法，而不是子类的 f()方法。IOR 变异算子重新命名了父类中被重写的方法，将调用子类改为调用父类。只有当子类重写的方法和父类的方法不同时，变异体才会被杀死。表 8-21 是一个简单的例子。

表 8-21　IOR 变异算子

原　程　序	变　异　体
class List()： … def m(self)： self. f() def f(self)： … class Stack(List)： def f(self)： …	class List()： … def m(self)： ➡ self. f1() ➡ def f1(self)： … class Stack(List)： def f(self)： …

7. ISD(删除 super 关键字)

子类可能会重写父类的变量或方法,子类的实例仍然可以通过关键字 super 来调用父类中隐藏的变量或方法。ISD 变异算子将 super 关键字删除,原本调用父类的隐藏变量改成了调用子类的变量。只有当子类的变量或者方法重写有意义时,变异体才会被杀死。表 8-22 是一个简单的例子。

表 8-22 ISD 变异算子

原 程 序	变 异 体

class Stack(List):
    …
    def Pop(self):
        return super(Stack,self).size
	class Stack(List):
	    …
	    def Pop(self):
	➡ return size

8. ISI(插入 super 关键字)

ISI 变异算子和 ISD 正好相反,它在没有 super 关键字的地方插入 super,原本调用子类的变量变为调用父类的隐藏变量。它和 ISD 变异算子产生的变异体一样,都只有当子类的变量或者方法重写有意义时,变异体才会被杀死。表 8-23 是一个简单的例子。

表 8-23 ISI 变异算子

原 程 序	变 异 体

class Stack(List):
    …
    def Pop(self):
        return super(Stack,self).size
	class Stack(List):
	    …
	    def Pop(self):
	➡ return size

8.3.3 变异测试的过程

变异测试的目的是为了检测测试用例的完备性,原程序和测试用例集是已有的,具体的过程如下。

(1) 根据原程序的特点,设定一系列的变异算子。

(2) 通过变异算子,在原程序的基础上生成变异体。

(3) 从变异体中识别等价变异体。

(4) 在原程序和非等价变异体上顺序执行测试用例(一旦有测试用例杀死该变异体就不再在该变异体上执行其他测试用例)。

(5) 检查被杀死的变异体个数,计算 KM 值(变异评分)。

(6) 如果达到要求的 KM 值(变异评分),则结束测试;否则增加新的测试用例,再回到步骤(4)。

对于步骤(6),实际测试中可以要求达到一定的 KM 值,也可以要求杀死所有的变异

```
(10)            high-=1
(11)        while low< high and array[high]< key:
(12)            array[low]= array[high]
(13)            low+=1
(14)            array[high]= array[low]
(15)        array[low]= key
(16)        return low
(17)    if __name__=='__main__':
(18)        n= input('Enter n: ')
(19)        array= [input('Enter array: ') for i in range(n)]
(20)        print "array before quicksort:",array
(21)        quick_sort(array,0,len(array)-1)
(22)    print  "array after quicksort:",array
```

已有的测试用例集如表 8-24 所示，其中，n 表示待排序数组的元素个数，array 表示待排序数组。

表 8-24　变异测试用例编号

用例编号	待排序数组	用例编号	待排序数组
1	n=1 array=[5]	5	n=2 array=[2,3]
2	n=2 array=[4,3]	6	n=6 array=[1,2,3,4,5,6]
3	n=5 array=[20,1,5,7,10]	7	n=5 array=[1,5,7,23,4,5]
4	n=7 array=[19,12,11,7,4,3,1]	8	n=9 array=[10,2,3,22,13,15,4,8,19]

快速排序程序中没有涉及面向对象的内容，类的变异算子对该程序没有任何影响，可以只选择使用传统的变异算子。因为传统的变异算子较多，限于篇幅仅选择 ASRS、AOR 和 LCR 三种，只考虑一阶变异，不考虑高阶变异。

根据选择的三种变异算子，对原程序进行变异，可以得到如表 8-25 所示的变异体，在表中，序号是变异序号，行号是原程序中的行号。

表 8-25　快速排序的部分变异示例

序号	变异算子	行号	原　程　序	变　异　体
1	ASRS	10	high -= 1	high+= 1
2	ASRS	10	high -= 1	high *= 1
3	ASRS	10	high -= 1	high/= 1
4	ASRS	10	high -= 1	high%= 1
5	ASRS	13	low += 1	low-= 1
6	ASRS	13	low += 1	low *= 1
7	ASRS	13	low += 1	low/= 1

必须充分考虑局部故障或者特殊事件对系统最终故障的影响。故障树用来描述故障的形成原因,基于故障树的软件测试方法用于测试事件处理是否正常。

8.4.1 故障树的概念

故障树是一种倒立树状的逻辑图像,逐层对故障形成的原因进行细化,通过表征符号将故障形成原因连接,最终形成一个因果关系图。故障树常用的基本符号主要包括事件和逻辑门,如表 8-27 所示。

表 8-27 故障树法常用的基本符号

名 称	符 号	含 义
顶事件		系统不希望发生的目标状态,位于故障树的顶端
中间事件		独立的事件,在顶事件和底事件之间,既可以是逻辑门的输入事件,也可以是逻辑门的输出事件
基本事件		已查明或者未查明但必须探明发生原因的底事件,在故障树的底端,是逻辑门的输出事件
未开展事件		无须探明发生原因的底事件
转移符号		使图形简明和防止重复画图的符号
与门		所有的输入事件发生,输出事件才发生
或门		至少有一个输入事件发生,输出事件才发生
异或门		当且仅当一个输入事件发生,输出事件才发生
优先与门		输入事件按规定从左到右顺序发生,输出事件才发生

图 8-20 是一个简单的故障树示例。

其中,T 是顶事件,G_1、G_2、G_3、G_4 和 G_5 是中间事件,X_1、X_2、X_3、X_4、X_5、X_6、X_7、X_8 和 X_9 是底事件。G_2 和 G_4 作为输出事件的是与门,T、G_1、G_3 和 G_5 作为输出事件的是或门。

8.4.2 故障树的建立和分析

故障树可以形象直观地显示软件系统存在的故障,并且可以清晰地显示各个故障之间的逻辑关系。通过故障树可以找到系统存在的一些潜在故障因素,方便对系统进行改

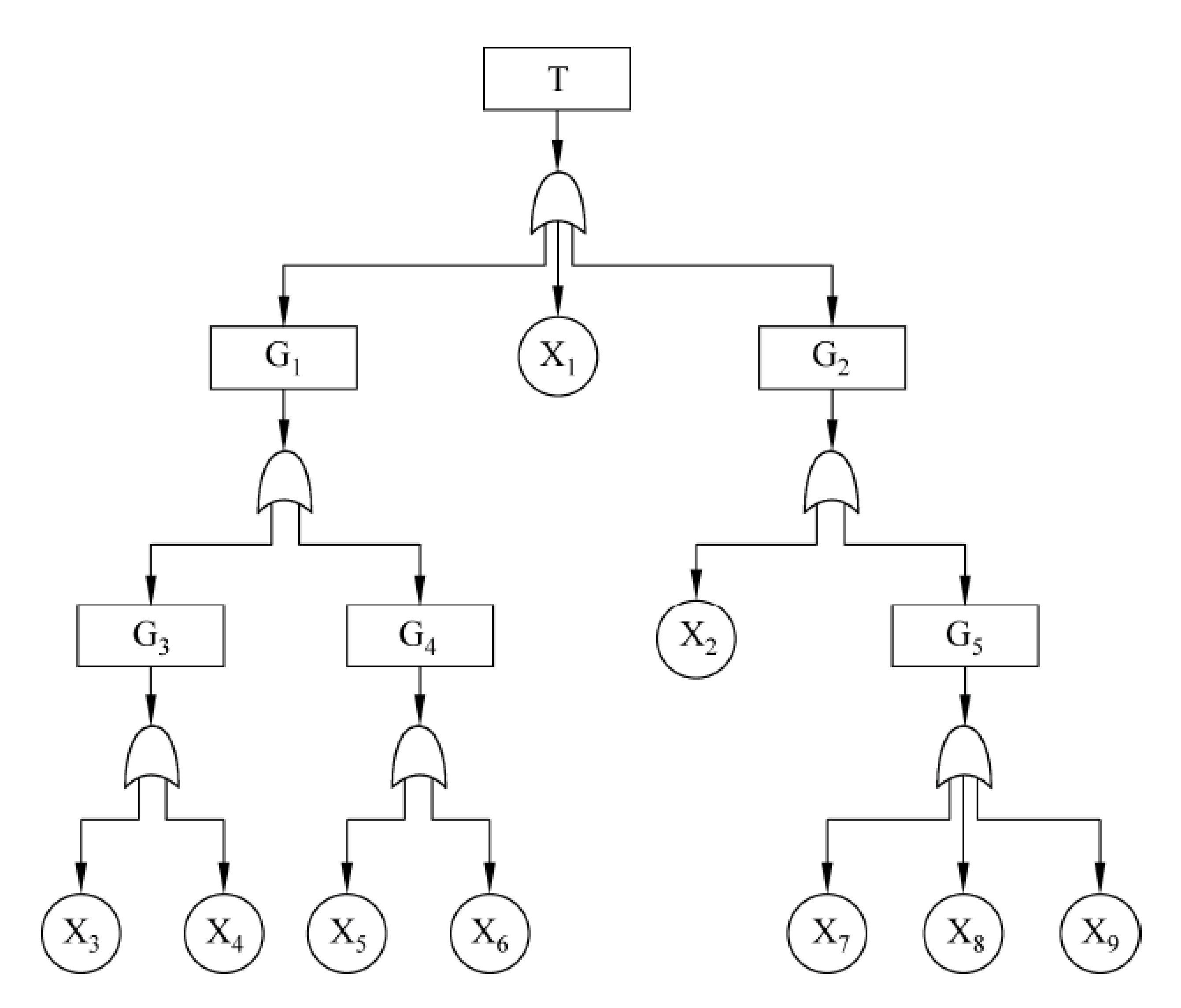


图 8-20 简单的故障树

进。故障树的建立十分重要，故障树是否完善、层次是否分明将直接关系到对故障树定量分析和定性分析的结果。

建立故障树的步骤如下。

(1) 分析系统可能存在的故障，确定故障树的范围。

(2) 确定故障树的顶事件，通常选择后果最严重的故障或者发生率最高的事件。

(3) 查找顶事件发生的原因(中间事件)，逐层查找故障发生原因直到无法找到发生原因的事件(底事件)。

(4) 利用逻辑门连接所有事件，完成故障树的构建。

建立故障树后可以对故障树进行定量分析和定性分析。定量分析是根据底事件发生的概率计算顶事件的发生概率。实际测试中可以不进行定量分析，只需要进行定性分析。定性分析主要是为了找出导致故障树顶事件的所有可能的故障模式，即找出故障树所有的最小割集。割集是底事件的集合，当集合中的所有底事件发生时，顶事件必然发生。如果去掉割集中的任意一个底事件就不再是割集，这样的割集就是最小割集。最小割集就是导致顶事件发生的不能再简化的割集。

最小割集的求解常用的两种方法是上行法和下行法。上行法是自下而上地进行计算。将故障树中的逻辑符号用集合符号代替，例如，或门用"∪"表示，与门用"∩"表示。再根据集合符号进行计算，得到每一层的集合表示，自下而上表示直到顶事件，集合全部用并集表示。

用上行法求解图 8-2 中的简单故障树，具体过程如下。

故障树第三层的表达式为：

$$G_3 = X_3 \cup X_4,\ G_4 = X_5 \cap X_6,\ G_5 = X_7 \cup X_8 \cup X_9$$

第二层的表达式为：

8.4.3 基于故障树的测试用例设计

故障树的定性分析可以找出最小割集，每个最小割集对应一种故障模式。可以根据每种故障模式设计测试用例来实现故障的检测。最小割集中有多个元素，每个元素作为输入有一定的输入范围。可以用黑盒测试中的等价类划分方法对每个最小割集的元素进行划分，再将各个元素的值进行组合测试。

例如，对图8-20中的基本故障树定性分析，得到最小割集为：$\{X_3\}$，$\{X_4\}$，$\{X_5, X_6\}$，$\{X_1\}$，$\{X_2, X_7\}$，$\{X_2, X_8\}$，$\{X_2, X_9\}$。对于最小割集$\{X_5, X_6\}$，假设等价类划分后的元素的取值为 X_5：a_1, a_2；X_6：b_1, b_2。对 X_5 和 X_6 进行组合得到的测试用例为：$\{a_1, b_1\}$、$\{a_1, b_2\}$、$\{a_2, b_1\}$和$\{a_2, b_2\}$。最后得到的用例个数为 $2\times2=4$。

故障树的测试用例设计过程如下。

(1) 分析系统存在的故障，画出故障树。

(2) 定性分析故障树，得到最小割集。

(3) 采用等价类划分最小割集中的每个元素，组合各个元素的取值作为测试用例的输入。

(4) 将所有最小割集的测试用例组成最终的测试用例集。

以电梯故障为例，常见的电梯故障有电梯开门故障、关门故障、电梯急停等。电梯急停是一个比较常见的故障，并且危害较大，现将对这一故障进行详细说明。

下面是故障以及引起故障的原因。

T：电梯急停

G_1：安全回路故障

G_2：急停信号

G_3：门锁故障

G_4：供电电源短路

G_5：变频器发出故障信号

G_6：门刀碰门锁

G_7：电流指令过大

G_8：有过电流信号

G_9：超载故障

X_1：超速

X_2：继电器接触不良

X_3：楼层位置计算有偏差

X_4：开关故障

X_5：有人扒门

X_6：绝缘老化

X_7：气象原因

X_8：有过电压信号

X_9：停在门区

X_{10}：瞬间断开

X_{11}：称重信号出错

X_{12}：晶体管模组击穿

X_{13}：晶体管模过电流

X_{14}：未处理超载信号

X_{15}：超负荷运行

这些事件的关系如下所示。

(1) G_1、X_1 和 G_2 中任意一个事件发生，T 就会发生。

(2) G_3、X_2 和 G_4 中任意一个事件发生，G_1 就会发生。

(3) X_3 和 G_5 中任意一个事件发生，G_2 就会发生。

(4) X_4、G_6 和 X_5 中任意一个事件发生，G_3 就会发生。

(5) X_6 和 X_7 中任意一个事件发生，G_4 就会发生。

(6) G_7、X_8 和 G_8 中任意一个事件发生，G_5 就会发生。

(7) X_9 和 G_{10} 两个事件都发生，G_6 才会发生。

(8) G_9 和 X_{11} 中任意一个事件发生，G_7 就会发生。

(9) X_{12} 和 X_{13} 中任意一个事件发生，G_8 就会发生。

(10) X_{14} 和 X_{15} 两个事件都发生，G_9 才会发生。

根据上面各个故障和故障之间的关系，可以得到如图 8-21 所示的故障树。

得到故障树后需要对故障树行定性分析，以求得最小割集。这里采用下行法，具体过程如下。

(1) 自上而下的第一层是顶事件，顶事件是作为或门的输出，因此将它的三个输入事件 G_1、X_1 和 G_2 写成三行。

(2) 第二层中有三个事件，从左到右依次处理，底事件不进行处理。G_1 是或门的输出，因此将它的三个输入事件 G_3、X_2 和 G_4 写成三行代替 G_1；G_2 是或门的输出，因此将它的两个输入事件 X_3 和 G_5 写成两行代替 G_2。

(3) 第三层中有五个事件，从左到右依次处理，底事件不进行处理。G_3 是或门的输出，因此将它的三个输入事件 X_4、G_6 和 X_5 写成三行代替 G_3；G_4 是或门的输出，因此将它的两个输入事件 X_6 和 X_7 写成两行代替 G_4；G_5 是或门的输出，因此将它的三个输入事件 G_7、X_8 和 G_8 写成三行代替 G_5。

(4) 第四层中有八个事件，从左到右依次处理，底事件不进行处理。G_6 是与门的输出，因此将它的两个输入事件 X_9 和 X_{10} 写成水平的一行代替 G_6；G_7 是或门的输出，因此将它的两个输入事件 G_9 和 X_{11} 写成两行代替 G_7；G_8 是或门的输出，因此将它的两个输入事件 X_{12} 和 X_{13} 写成两行代替 G_8。

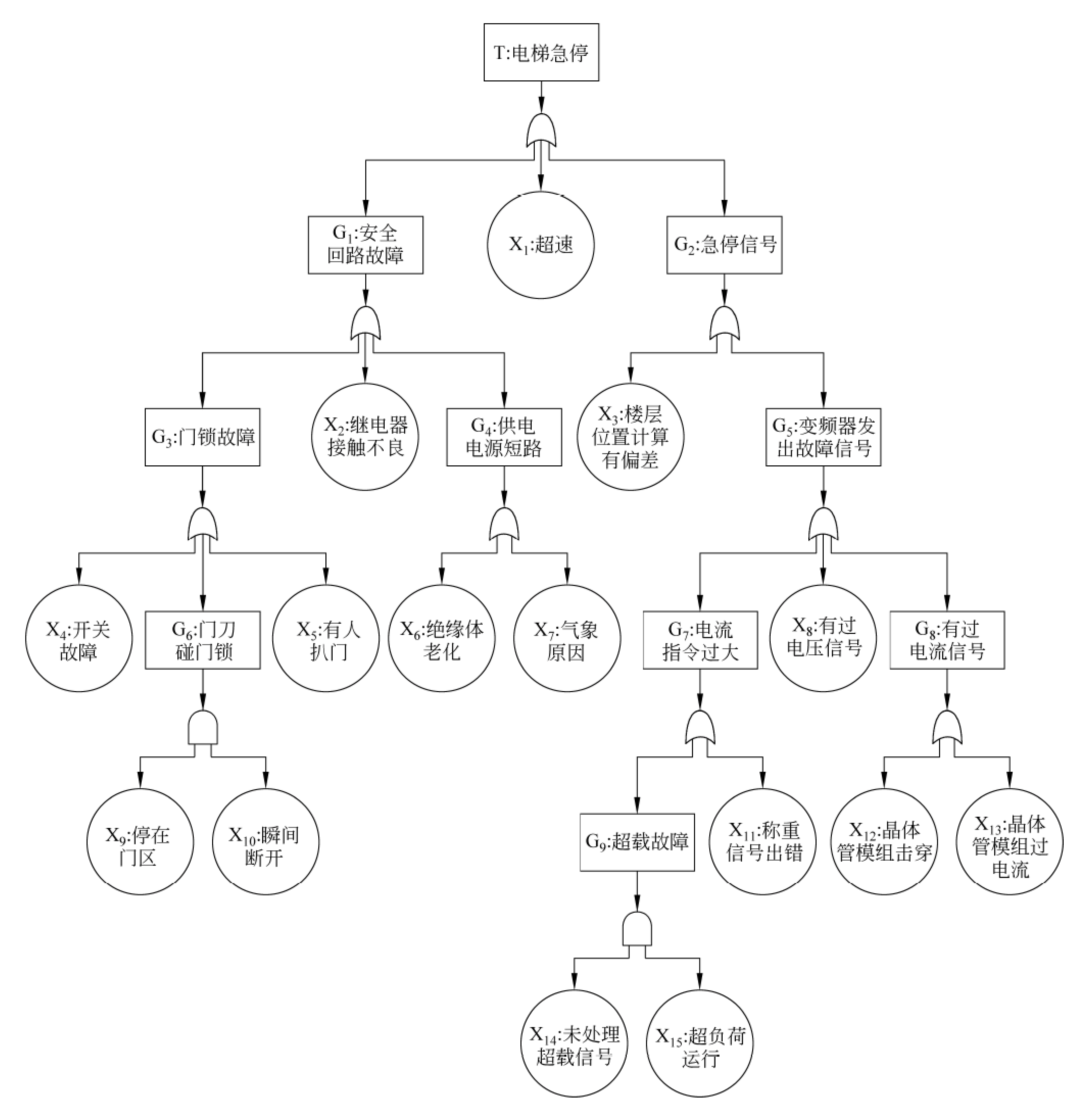


图 8-21 电梯急停故障树

(5) 第五层中有六个事件,从左到右依次处理,底事件不进行处理。G_9 是与门的输出,因此将它的两个输入事件 X_{14} 和 X_{15} 写成水平的一行代替 G_9。

(6) 第六层都是底事件,不再进行处理,计算结束。

根据上述过程得到的下行法表如表 8-29 所示。

最终得到的最小割集为:$\{X_4\}$,$\{X_9, X_{10}\}$,$\{X_5\}$,$\{X_2\}$,$\{X_6\}$,$\{X_7\}$,$\{X_1\}$,$\{X_3\}$,$\{X_1\}$,$\{X_{14}, X_{15}\}$,$\{X_{11}\}$,$\{X_8\}$,$\{X_{12}\}$,$\{X_{13}\}$。最小割集中的每个元素都只取一个典型情况,组合各个元素的取值作为测试用例的输入。相当于每个割集对应一个测试用例,如表 8-30 所示。

附　　录

附录 A　软件测试大事记

从第一个软件缺陷出现开始，至今将近七十年的历史，在这个发展过程中，软件测试一直伴随着软件技术以及软件工程的发展而发展。实际上，软件测试本身的发展也是软件工程发展的一个重要体现，本附录收集了软件测试历史上的重要事件，供读者参考。

1947 年，Mark Ⅱ计算机遇到一个飞蛾，这是历史上第一个计算机软件缺陷。

1957 年，Charles L. Baker 在给 Dan McCracken 所著的《数字计算机程序》(*Digital Computer Programming*)一书的评论中，首次区分了软件测试和软件调试。

1958 年，Gerald M. Weinberg 成立世界上第一个独立的测试团队，该团队为火星计划(Mercury Project)的操作系统开发提供独立的测试。

1959 年，D. G. Malcolm，J. H. Roseboom，C. E. Clark 和 W. Fazar 等人在其经典论文《程序评估和评审技巧》中介绍了在分析给定项目计算任务时所用的工作分解结构(Work Breakdown Structure，WBS)方法。

1961 年，Gerald Weinberg and Herbert Leeds 在其著作《计算机程序基础》(*Computer Programming Fundamentals*)中包含一章关于软件测试的内容，指出软件测试应该证明计算机程序的适合性(adaptability)而不是处理信息的能力。同年，在 IBM 工作的 Burton Grad 提出了决策表方法。

1967 年，在 IBM 白皮书《控制程序的功能测试评价》中，William Elmendorf 要求严格的软件测试方法。

1968 年，北大西洋公约 NATO 支持的软件工程会议中，提出问题：软件测试是否能够确保对客户而言除了匹配功能规格说明书外，其是否对客户而言是最有用的。

在 1968 年软件工程会议上，已经开始重新思考软件测试的目的。Friedrich Ludwig Bauer 在 NATO 的软件工程会议上首先提出了软件危机(Software Crisis)一词，意思是指编写正确、可理解的和可验证的计算机程序的困难。

1969 年，Edsger Dijkstra 在意大利罗马召开的 NATO 支持的会议上指出，软件只能表明软件缺陷的存在，而不能表明没有缺陷。

1971 年，Richard Lipton 在其论文《计算机程序的故障诊断》(*Fault Diagnosis Computer Programms*)中提出了变异测试(Mutation Testing)的概念。

1972 年，Bill Hetzel 在 North Carllina 大学举行第一次以软件测试为主题的正式议会(1972.6.21—1972.6.23)，在该会议上，出版书籍《程序测试方法》，主要聚焦于软件测试和确认。

1972 年，Edsger Dijkstra 在其题为“卑微的程序员”(Humble Programmer)的 ACM 图灵讲座中指出：不应该是先编写程序，然后证明程序的正确性。程序正确性的证明的

1996 年，伊利诺斯理工学院发布了软件测试成熟度模型（The Testing Maturity Model，TMM）。

1996 年，David Cohen，Siddhartha Dalal，Jesse Parelius 和 Gardner Patton 在其论文《自动测试生成-组合设计方法》（*The Combinatorial Design Approach to Automatic Test Generation*）中提出了成对组合测试。

2002 年，Kent Beck 在其出版书籍《案例说测试驱动开发》中提出了一种新的软件开发技巧，在软件功能编码前，先编写测试用例，也就是测试驱动开发。

2008 年，eo van der Aalst 在西北太平洋软件质量会议上，提出了软件测试即服务的概念（Software Testing as a Service，STaaS）。STaaS 是将软件测试用于通过互联网提供给客户的一个软件测试模型。

附录 B　常见正交测试表

本附录摘取了部分正交表，供读者在测试过程中参考。

1. $L_4(2^3)$

0	0	0
0	1	1
1	0	1
1	1	0

2. $L_9(3^4)$

0	0	0	0
0	1	2	1
0	2	1	2
1	0	2	2
1	1	1	0
1	2	0	1
2	0	1	1
2	1	0	2
2	2	2	0

3. $L_{12}(2^4 \times 3^1)$

0	0	0	0	0
0	0	1	1	1
0	0	1	1	2
0	1	0	0	2
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	0	1	2
1	0	1	0	0
1	1	0	1	1
1	1	1	0	2
1	1	1	1	0

4. $L_8(2^4 \times 4)$

0	0	0	0	0
0	0	1	1	2
0	1	0	1	1
0	1	1	0	3
1	0	0	1	3
1	0	1	0	1
1	1	0	0	2
1	1	1	1	0

5. $L_{12}(2^{11})$

0	0	0	1	0	0	1	0	1	1	1
0	0	1	0	0	1	0	1	1	1	0
0	0	1	0	1	1	1	0	0	0	1
0	1	0	0	1	0	1	1	1	0	0
0	1	0	1	1	1	0	0	0	1	0
0	1	1	1	0	0	0	1	0	0	1
1	0	0	0	1	0	0	1	0	1	1
1	0	0	1	0	1	1	1	0	0	0
1	0	1	1	1	0	0	0	1	0	0
1	1	0	0	0	1	0	0	1	0	1
1	1	1	0	0	0	1	0	0	1	0
1	1	1	1	1	1	1	1	1	1	1

6. $L_{12}(2^2\times 6^1)$

0	0	0
0	0	2
0	0	4
0	1	1
0	1	3
0	1	5
1	0	1
1	0	3
1	0	5
1	1	0
1	1	2
1	1	4

7. $L_{16}(2^8\times 8)$

0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	4
0	0	1	1	0	0	1	1	2
0	0	1	1	1	1	0	0	6
0	1	0	1	0	1	0	1	1
0	1	0	1	1	0	1	0	5
0	1	1	0	0	1	1	0	3
0	1	1	0	1	0	0	1	7
1	0	0	1	0	1	1	0	7
1	0	0	1	1	0	0	1	3
1	0	1	0	0	1	0	1	5
1	0	1	0	1	0	1	0	1
1	1	0	0	0	0	1	1	6
1	1	0	0	1	1	0	0	2
1	1	1	1	0	0	0	0	4
1	1	1	1	1	1	1	1	0

8. $L_{18}(3^6\times 6)$

0	0	0	0	0	0	0
0	0	1	1	2	2	1
0	1	0	2	2	1	2
0	1	2	0	1	2	3
0	2	1	2	1	0	4
0	2	2	1	0	1	5
1	0	0	2	1	2	5
1	0	2	0	2	1	4
1	1	1	1	1	1	0
1	1	2	2	0	0	1
1	2	0	1	2	0	3
1	2	1	0	0	2	2
2	0	1	2	0	1	3
2	0	2	1	1	0	2
2	1	0	1	0	2	4
2	1	1	0	2	0	5
2	2	0	0	1	1	1
2	2	2	2	2	2	0

13. $L_{25}(5^6)$

0	0	0	0	0	0
0	1	1	1	1	1
0	2	2	2	2	2
0	3	3	3	3	3
0	4	4	4	4	4
1	0	1	2	3	4
1	1	2	3	4	0
1	2	3	4	0	1
1	3	4	0	1	2
1	4	0	1	2	3
2	0	2	4	1	3
2	1	3	0	2	4
2	2	4	1	3	0
2	3	0	2	4	1
2	4	1	3	0	2
3	0	3	1	4	2
3	1	4	2	0	3
3	2	0	3	1	4
3	3	1	4	2	0
3	4	2	0	3	1
4	0	4	3	2	1
4	1	0	4	3	2
4	2	1	0	4	3
4	3	2	1	0	4
4	4	3	2	1	0

附录 C　PICT 工具指南

PICT(Pairwise Independent Combinatorial Testing tool)是微软公司推出的组合测试工具。目前版本为 3.3,下载地址为 http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi。下载以后,直接安装,其过程比较简单。在安装完成以后,单击“开始”,在“运行”框中输入“CMD”,进入命令行窗口。在命令行提示符下执行 pict:

```
C:\Users\clz> pict
```

如果看到以下信息,表示安装成功。

```
Pairwise Independent Combinatorial Testing
```

```
py.test --maxfail=N                #N次失败之后停止执行
```

执行选择用例的命令如下。

```
py.test test_mod.py                #执行模块中的测试用例
py.testsomepath                    #执行路径 somepath 中的测试用例
py.test -k stringexpr              #执行字符串表达式 stringexpr 中的测试用例
```

3. pytest 脚本自动发现

测试脚本的自动发现,pytest 不需要提供单独的测试列表,它可以自动搜索项目的源代码树,寻找项目的所有测试。pytest 从整个项目的基目录开始搜索;如果要测试名为 demo 的包,pytest 会从 demo 的父目录开始搜索测试。然后 pytest 自动向下递归地搜索项目的每个子目录。若将 pytest 编写的测试放在非包目录中,应保持名称是唯一的。在每一个目录中,pytest 总是寻找名称以'test_开头或以_test 结尾的 Python 模块。

一般地,在组织项目结构时,可以采用两种不同的方式。

方式一:外部测试目录,将测试代码和软件代码完全分开。Python 中,每个 py 文件被称为模块,每个具有__init__.py文件的目录被称为包。在 mypkg 目录中,必须存在一个__init__.py 文件,即使这个文件为空也必须存在。

```
setup.py #  your distutils/setuptools Python package metadata
mypkg/
    __init__.py
    appmodule.py
    ...
tests/
    test_app.py
    ...
```

方式二:内嵌测试目录,将测试代码存放在软件代码目录下,作为一个单独的目录存在,在这种方式中,测试代码将随着软件代码一同发布。

```
setup.py #  your distutils/setuptools Python package metadata
mypkg/
    __init__.py
    appmodule.py
    ...
    tests/
        test_app.py
        ...
```

无论采用何种模式,都可以采用下面的几种模式之一执行测试,pytest 都会根据前面所描述的自动发现机制,寻找存在的测试用例并自动执行。

```
py.test tests/test_app.py          #执行外部测试目录中的测试
py.test mypkg/test/test_app.py     #执行内嵌测试目录中的测试
```

```
===========================test session starts =====================
platform win32 --  Python 2.7.3 --  py-1.4.26 --  pytest- 2.6.4
collected 1 items

tests/test_sample1.py.
=========================1 passed in 0.02 seconds ===================
```

如果测试失败，pytest 将给出详细的失败信息。若将 test_sample1. py 中 func 的预期输出修改为 5，如测试 test_sample2. py 所示，那么将导致测试失败。

```
#test_sample2.py
import  pytest
from app.sample2 import func

def test_answer():
    assert func(3)==5
```

在目录 e:\pythonstudy\pytestdemo1 下执行：

```
E:\pythonstudy\pytestdemo1>py.test
```

其测试结果如下。

```
===========================test session starts===================
platform win32--Python 2.7.3--py-1.4.26--pytest-2.6.4
collected 2 items

tests/test_sample1.py .
tests/test_sample2.py F

=============================FAILURES============================
_______________________________ test_answer _____________________________

    def test_answer():
>       assert func(3)==5
E       assert 4==5
E        +  where 4=func(3)

tests\test_sample2.py:5: AssertionError
================1 failed, 1 passed in 0.05 seconds===================
```

5. 参数化及其他

在前面的讨论中，一个测试函数仅包含一个测试用例，有多少个测试用例必须拥有相同数目的测试函数，这种情况会造成大量相同的测试代码，也不利于阅读测试代码。pytest 提供了参数化 fixture，也就是@pytest. mark. parametrize 来实现一个测试函数包含多个测试用例。

```
#pytestdemo2/tests/test_sample2.py
import  pytest
from app.sample1 import add

@pytest.mark.parametrize("x,y,r", [
    (3,5, 8),
    (2,4, 6),
    (6,9, 12),
])
def test_answer(x,y,r):
    assert add(x,y)==r
```

执行以下命令：

```
E:\pythonstudy\pytestdemo2>py.test tests/test_sample2.py
```

得到如下结果。

```
============================ FAILURES ============================
________________________ test_answer[6-9-12] ________________________

x=6, y=9, r=12

    @pytest.mark.parametrize("x,y,r", [
        (3,5, 8),
        (2,4, 6),
        (6,9, 12),
    ])
    def test_answer(x,y,r):
>       assert add(x,y)==r
E       assert 15==12
E        +  where 15=add(6, 9)

tests\test_sample2.py:12: AssertionError
=================== 1 failed, 2 passed in 0.04 seconds ===================
```

实际上，已经知道 6+9=15，所以这个测试用例必定失败，可以在参数化过程中加以标注。将 test_sample2.py 修改为 test_sample3.py。其内容如下。

```
#pytestdemo2/tests/test_sample2.py
import  pytest
from app.sample1 import add

@pytest.mark.parametrize("x,y,r", [
    (3,5, 8),
    (2,4, 6),
```

```
pytest.mark.xfail( (6,9, 12)),
])
def test_answer(x,y,r):
    assert add(x,y)==r
```

执行以下命令：

```
E:\pythonstudy\pytestdemo2>py.test tests/test_sample3.py
```

得到如下结果。

```
=========================== test session starts====================
platform win32- - Python 2.7.3- - py- 1.4.26- - pytest- 2.6.4
collected 3 items

tests/test_sample3.py ..x

=================== 2 passed, 1 xfailed in 0.06 seconds================
```

附录 E　最长公共子序列示例

最长公共子序列(Longest Common Subsequence,LCS)的定义是,一个序列 S,如果分别是两个或多个已知序列的子序列,且是所有符合此条件序列中最长的,则 S 称为已知序列的最长公共子序列。假设序列为下面两个序列,如果要人工验证给出其预期的结果,其困难是非常巨大的。

wxfbfckktkamlldlgqzphmhpcbivvlvzwrumcgflcplvetogezeorzniqpphglyswiigbufsjnar
ejemjziixusbbfzfxnefltlckriwcntgcjkuunladgtsdjqkwvwwzkwnolsybdknavrmllfuihuofhxez
rnskedvsfhofxyqaqdrszhsjfbehbfagbvpndxmymfpbplvhvgiutbeivmeraxspghhdokktprwtnm
zuhwqfapuschcuserhcnsfkzsmeoskkpmfmoioczjqohovghkxbidldyxjattwlldlkldlzwierxurjir
eiwtdrrfgunskdxsgdthvwegjrengrculnuksmfqtrvonulbdcycrazcghchfabghdfuphjhxisdofew
tkojsdzkomzfqozjfztnfvieqlfouineiiktxeybzdiijakunugiwufldxnz

khkeanikunhhqkcojlvzzbascuuzwqbnjwatgrgglulcubytcnwofrzpieqdipkbhthpruqtqitiy
hmxfkuoxenlzhpmaeodpdbnllgouslxczcmsjmqnfoqqttsbtrgwyxgdxfgvhxwrqlnkzjfnssirtm
qdwkiyhhdpyljprysknwxlriwrzxqwbocewycchyxzaswxuhdjvmityixhwumoqhajbodvvihkvz
unlorpnafnbnbkgsyelujclubfzimqilcjjkqdpsotczdevcmzvifjgsckkehiryoiwphqwaslktdimnzs
pqhigicarinifvnsjfjrgbqdoosfnenjskrqyczzmctwwhkcscbehbgcjqomavgccpmzwycmkymfihg
nxmqomaapjwjtjfebnuwczqnlmeigtiwaegkdzrhdwyqficmihmtnsvvowbvebnbmbtyrzvylkof
gpxdkpdsfcythhbdfctgeqirxscjkeluaxwdsgprmyrdgzmobwzrzjljgmkkalvmdospgkdhkmfdjf
jysufvwdddcyyfbjldbzcvbmnnucaxjfkyrdulfffrkfkiwvopyncejjubtxtagobtcpvoyhwbetuarox
mydzkhlhnjlsgsdrpekofwbvwjvaflzbvfujbkviitpuoklnfysyajslhaebmzwndgiiyisahvhvazwm
urlmjvloxucauoqhkgqzuqgpddgyxbjxghxcghdoaluawilbectxiyebxocbhakgxybikvpwbphux
ncjdwnfvjzedntslnzpcbkxsebeeuqdmoodkmpdmtifytpkhiahjscjfatekshovvzuaihbkfzclchxix

dljrfxwgaumnhhdnqcbidgjmttilrpkwaxbjaemefuvcolkqbnknpcnyikdwbnuvbqotzgvsnlgziij
noqqdcyenzfzwufguuzxbqzbdhznkmpotdgtzrqplgqlvpwqodxebabmyexzvxysazoujxrzvamcn
ypflxlpokkmscxdxggsapuyxcfcosrooeexygtkiphaxgcummbrsjbcnqwsekqodpijzhxenmjxqks
rgxnkbhnwqmeooahoyxkjnhvtgptirpfokiwxctfyntjukwaoyfmlcwitceanggrofwpioepydocqb
penmdsqfgbngipyuixnnermstfbafexbbfbjspguutwsltqvlaqwzsjpzpkzovvcghaanhzucqccjgba
ypdqtachbktgrxmgxcfbzsyaqziphvxkmxdtwpfdzcrziybglxhuvmbnkviemjlihskrkuzafbczbsg
cbxekgryzfgkloiwemrukjojzjrfiplnxkmhambxinngaeuunwgctojjkvosozgfjqcneusmnojslehq
hztjzsxvmuakez

oriented software applications,2005[C]. IEEE.
[46] Wu Y,Chen M,Offutt J. uml based integration testing for component based software[J]. Lecture Notes in Computer Science,2003,2580:251-260.
[47] 陈翔,顾庆.变异测试:原理、优化和应用[J].计算机科学与探索,2012(12):1057-1075.
[48] 茆亮亮.变异测试技术应用研究[D].中国科学技术大学,2010.
[49] 黄陇,杨宇航,李虎.参数配对及 n-way 组合覆盖算法研究[J].计算机学报,2012(02):2257-2269.
[50] 陈蕊.程序中不可达路径的识别及其在结构测试中的应用[D].中国科学院研究生院(计算技术研究所),2006.
[51] 邹伟.故障模型在软件测试上的应用探讨[J].软件导刊,2008(11):35-36.
[52] 漆莲芝,张军,谢敏.故障树分析测试用例生成技术研究与应用[J].信息与电子工程,2010(05):594-597.
[53] 冯文祥,刘万军,邓月,等.故障树分析技术在软件测试中的研究[J].计算机测量与控制,2010(08):1764-1767.
[54] 刘文红,王占武,吴欣.故障树分析技术在软件测试中的应用[J].系统工程与电子技术,2004(07):985-987.
[55] 吴蔚峰.故障树理论在电梯故障诊断中的应用[J].科学与财富,2014(12):241-242.
[56] 周妍.故障树自动生成系统的研究与开发[D].大连理工大学,2005.
[57] 吴鹏,施小纯,唐江峻,等.关于蜕变测试和特殊用例测试的实例研究[J].软件学报,2005(07):1210-1220.
[58] 王冠,景小宁,王彦军.基本路径测试中的 McCabe 算法改进与应用[J].哈尔滨理工大学学报,2010(01):48-51.
[59] 蔡立志.基于 CPN 状态空间的软件场景测试[J].计算机应用与软件,2010(09):37-40.
[60] 刘攀,缪淮扣,曾红卫,等.基于 FSM 的测试理论、方法及评估[J].计算机学报,2011,34(6):965-984.
[61] 刘攀.基于 FSM 的测试用例生成和测试优化[D].上海大学,2011.
[62] 王子元,钱巨,陈林,等.基于 One-test-at-a-time 策略的可变力度组合测试用例生成方法[J].计算机学报,2012(12):2541-2552.
[63] 方贤文,赵艳,殷志祥.基于 Petri 网的软件测试分析[J].计算机技术与发展,2007(02):96-98.
[64] 林红昌,胡觉亮,丁佐华.基于 Petri 网的软件测试用例的产生及分析[J].计算机工程与应用,2009(17):57-60.
[65] 占学德.基于 UML statecharts 测试方法的研究[D].上海大学,2005.
[66] 张保国.基于 UML Statechart 图的软件测试用例自动生成技术研究[D].湖南大学,2004.
[67] 解楠.基于 UML 活动图模型的测试用例自动生成方法研究[D].西安电子科技大学,2011.
[68] 李云平.基于 UML 活动图生成测试用例的研究[D].江苏大学,2011.
[69] 王志坚,金春.基于 UML 活动图生成系统测试场景的方法[J].计算机系统应用,2010(04):185-188.
[70] 刘青香.基于 UML 交互概览图的测试方法研究[D].重庆大学,2013.
[71] 逄瑞娟.基于 UML 顺序图的场景测试用例生成研究[D].青岛大学,2007.
[72] 陈雪清.基于 UML 顺序图的集成测试用例生成[D].吉林大学,2006.
[73] 陈雷.基于 UML 协作图的测试序列生成方法研究[D].河北工程大学,2011.
[74] 林鹿堃.基于 UML 序列图模型的软件测试研究[D].西安电子科技大学,2012.

[75] 米海容. 基于 UML 状态图的测试用例自动生成方法研究[D]. 西安电子科技大学,2011.
[76] 杜元柱,黄松,惠战伟,等. 基于变异分析的蜕变测试充分性条件[J]. 计算机应用,2014(S1):280-283.
[77] 陈蕾蕾. 基于等价类划分的蜕变测试方法优化[D]. 华东理工大学,2014.
[78] 张宇,张波,王俊杰,等. 基于二叉树满足 MC/DC 测试用例设计方法[J]. 微计算机信息,2010(03):171-173.
[79] 张卫祥,刘文红. 基于故障树分析与组合测试的测试用例生成方法[J]. 计算机科学,2014(S2):375-378.
[80] 潘丽丽,邹北骥,王天锷,等. 基于关键分支的不可行路径确定方法[J]. 北京工业大学学报,2010(05):716-720.
[81] 耿基鑫. 基于环形 FSCS 的 MT 研究[D]. 华东师范大学,2011.
[82] 张涌,钱乐秋,王渊峰. 基于扩展有限状态机测试中测试输入数据自动选取的研究[J]. 计算机学报,2003(10):1295-1303.
[83] 王建国,吴建平. 基于扩展有限状态机的协议测试集生成研究[J]. 软件学报,2001(08):1197-1204.
[84] 张涌,钱乐秋,王渊峰. 基于确定有限状态机的测试输入序列选取[J]. 计算机研究与发展,2002,39(9):1144-1150.
[85] 黄孝伦. 基于图的 MC/DC 最小测试用例集快速生成算法[J]. 计算机系统应用,2012(11):145-148.
[86] 张晶,胡学钢,张斌. 基于蜕变关系的聚类程序测试方法[J]. 电子测量与仪器学报,2011(08):688-694.
[87] 柳伟. 基于形式化 UML 测试序列生成方法研究[D]. 哈尔滨工程大学,2011.
[88] 王钊,白晓颖,戴桂兰. 基于有色 Petri 网模型的 GUI 测试用例自动生成技术[J]. 清华大学学报(自然科学版),2008(04):600-603.
[89] 赵保华,钱兰,周颢,等. 基于有限状态机的错误诊断算法[J]. 电子与信息学报,2006(09):1679-1683.
[90] 董蕾. 基于正交对的分布式软件测试研究与实现[D]. 苏州大学,2005.
[91] 周吴杰,张德平,徐宝文. 基于组合测试的软件故障定位的自适应算法[J]. 计算机学报,2011(08):1509-1518.
[92] 吴勋. 基于组合匹配的成对组合测试数据生成[D]. 湖南大学,2009.
[93] 张卫民,陈宏敏. 几种特定判定形式的 MC/DC 评估分析[J]. 飞行器测控学报,2008,27(4):56-60.
[94] Kaner Cem,JackFalk,Ngyen Hung Quo. 计算机软件测试[G]. 王峰,陈杰,喻琳,译. 北京:机械工业出版社,2004.
[95] 朱劼. 具有优先级的参数配对组合覆盖测试集生成策略的研究[D]. 上海师范大学,2008.
[96] 周吴杰,张德平,徐宝文. 快速生成两两组合测试用例集算法[J]. 东南大学学报(自然科学版),2011(05):943-948.
[97] 马秀飞,高翔,梅杓春. 两种基于 UIO 序列的测试序列生成算法及比较[J]. 计算机工程与应用,2005(22):76-79.
[98] 张广梅,李晓维,韩丛英. 路径测试中基本路径集的自动生成[J]. 计算机工程,2007(22):195-197.
[99] 张岩. 路径覆盖测试数据进化生成理论与方法[D]. 中国矿业大学,2012.

[125] 曾劲涛,陈建明.有参数约束的两两组合覆盖测试用例生成的研究[J].苏州大学学报(自然科学版),2008(01):45-49.
[126] 严俊,张健.组合测试:原理与方法[J].软件学报,2009(06):1393-1405.
[127] 陈翔.组合测试技术及应用研究[D].南京大学,2011.
[128] 陈翔,顾庆,王新平,等.组合测试研究进展[J].计算机科学,2010(03):1-5.
[129] 王子元,徐宝文,聂长海.组合测试用例生成技术[J].计算机科学与探索,2008(06):571-588.